A MULTI-TASKING OPERATING SYSTEM

FOR MICROCOMPUTERS

by

Roger Farrington Powell

A Thesis Submitted to the Faculty of the

DEPARTMENT OF ELECTRICAL ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

MASTER OF SCIENCE

In the Graduate College

THE UNIVERSITY OF ARIZONA

1 9 8 2

STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Roger Farrington Powell

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

Theodore B Williams
T. B. Williams
Professor of Electrical Engineering

December 4, 1982
Date

PREFACE

Certain terms and abbreviations that appear in this document are not standard English but are widely used in this field of study. These terms are not explained when first introduced. Other terms given special meaning in the thesis are defined when first introduced. The reader should be advised that there is a glossary at the end of the document, just before the list of references.

I wish to thank Dr. Theodore Bennett Williams of the University of Arizona, for his advice, guidance, and patience. It is no exaggeration to say that without him, this document would never have come to pass.

I wish also to extend thanks to Mr. Stanley Telson, now of Hewlett-Packard Corporation, for his valuable contributions, particularly in the design of the MACP macroprocessor.

This document was prepared and printed by word processing facilities at the University of Arizona Electrical Engineering Department's Microprocessor Laboratory.

TABLE OF CONTENTS

ABSTRACT

Ideas for a multi-tasking operating system along with a specific implementation are presented. The implementation was required to have file-system compatibility with an existing single-tasking operating system, and was required to run on hardware characterized by low-speed secondary storage and lack of relocation or context-switching capability. Much of the design effort was intended to overcome these limitations.

Intended applications are primarily single-user, but this is never a specific constraint. Subjective design goals were to make the system "attractive" to users by masking undesirable hardware characteristics, and to be easily used as an educational example.

Key Words and Phrases: microcomputer operating systems, multi-tasking, dynamically declared devices, minimal kernel.

CHAPTER 1

INTRODUCTION

The 1970's were the decade during which the microprocessor came out of the development laboratories and revolutionized the field of computer applications. One of the byproducts of this revolution has been the gradual shift in direction of software development support. In the past, necessity dictated timesharing of the expensive and highly powerful CPU units in the large mainframes. The low cost of the microprocessor has started to change that. The concept of one CPU per user is no longer necessarily wasteful.

The early 1980's have seen more exciting developments. The price of low-latency secondary storage has dropped considerably, threatening the floppy disk's domination of the microsystem market. In addition, more powerful processors such as the Motorola 68000 and the National 16000 provide both greatly increased processing power and the capability of memory management.

Systems using these components are increasingly limited by inadequate, single-tasking operating systems. This suggests the natural progression to multi-tasking. On a single user system, multi-tasking could allow, for example, text editing on one data file while another was being assembled or compiled. This particular example seems especially

appropriate in light of its applicability and the excellent mix. The text editing will undoubtedly be I/O-bound on the keyboard.

Another virtue of multi-tasking is the ability to add asynchronous "network driver" processes. This integrates CPU to CPU communication in the background and allows configurations where the microsystem can gracefully take advantage of non-local resources, such as large secondary memory, with a minimum of wasted user time.

All of the advantages of multi-tasking listed above centered around an implied theme: The primary design goal in a microsystem should be efficient use of the user's time. The rapid decrease in hardware costs are even more dramatic when compared to the steady increase of personnel cost. This also implies that making the system "pleasant" to use (this is, admittedly, subjective) should also be a design goal.

This paper describes the successful completion of a project at the University of Arizona, begun in 1978. This project anticipated the hardware developments referred to earlier, and comprised three basic parts:

1) Development of a specification for an operating system embodying the design concepts presented

2) Implementation of a prototype of such an operating system on available hardware, including the system itself, basic user programs (command interpreter,

debugger), and additional software to maintain compatibility with an older system on the same hardware

3) Development of needed software tools to produce item number 2

The author had sole responsibility for items 1 and 2, and partial responsibility for item 3.

The prototype system had to be implemented on an existing microcomputer system with a single floppy disk for secondary storage. The most vexing problem was the lack of any relocation or context-switching hardware. This fact, along with the special importance of certain memory areas to the processor, was the major "stumbling block" to multi-tasking. The entire design of the implementation was centered around overcoming these limitations. It is precisely this that makes this implementation interesting and different from typical small multi-tasking systems, which have either the benefit of address relocation hardware, context-switched memory, or fast secondary storage (for swapping). It will be shown that a pleasant and productive software-development environment was still provided even without those desirable features.

The result of the project is the QUAD3 operating system. Chapter 2 presents the tools developed to implement QUAD3 and Chapter 3 presents an overview of the system. Further chapters then expand upon the details of implementation.

In the rest of this chapter, the system that QUAD3 is based on, along with QUAD3's predecessor software will be presented. The processor and its limitations are presented to provide the necessary background. The ROM-based debugging monitor is presented since QUAD3 runs "on top" of it. The QUAD2 system is presented for three reasons: The older system contains many of the ideas that make up QUAD3, the drawbacks of QUAD2 inspired some of the features of QUAD3, and the issue of "QUAD2 compatibility" consistently forced the design of the newer system to be flexible enough to allow programs and files from the elder system to maintain their usefulness.

The 6502 Microprocessor

Our available CPU was the 6502 microprocessor. At the time of this writing, the 6502 was the world's best-selling microprocessor chip. It featured more addressing modes than either the 6800 or 8080, the comparable contemporaries, and a pipelined architecture that provided some speed advantages as well. Its popularity was primarily due to its low cost, thus high-volume applications such as commercial video games often chose the 6502.

The 6502 is primitive compared with chips now available. Some of the 6502's drawbacks are that the addressing modes are not available for all instructions, most registers cannot be pushed on the stack, short instructions to clear

or increment the accumulator are missing, and there is no 16 bit register. The most serious drawback is the special attention given low memory. In the 6502, memory is divided into 256 pages of 256 bytes each. In order to use two of the most powerful addressing modes, indexed-indirect and indirect-indexed, the indirection pointer must reside on page 0. In addition, the hardware stack pointer is only 8 bits wide; the stack resides exclusively on page 1. Another annoying drawback is that execution of an illegal instruction grinds the processor to a halt which only a hardware reset will cure. This, of course, destroys any status information which would serve to help identify the problem.

Other features of the 6502: It uses memory-mapped I/O, it has no explicit I/O instructions. Addresses on page 0 are kept with the least significant byte in the lower of two memory locations. This address format is maintained when return addresses are pushed on the hardware stack. Both maskable and non-maskable interrupts are available as well as a software interrupt instruction. There is a special bit in the condition register which changes the add and subtract instructions to decimal mode.

GOLEM Hardware

The GOLEM (General, Omnipotent, Lugubriously Encephalic Microsystem) physical system allows a variety of

hardware configurations. The minimum system needed for QUAD3 is listed here.

The CPU board contains several interval timers set up to interrupt on the NMI (Non-Maskable Interrupt) line. There is also a status register which may be set to provide a single-stepping facility. When enabled, this facility provides an NMI after the execution of each instruction below a certain address. The protected address region may be changed with jumpers. There are keys to send hardware reset, and NMI (called the STOP key) to the processor.

The serial I/O board provides twin UARTs with programmable baud rates. Another board contains the ROM for the firmware debugger, along with some RAM. The bulk of the RAM resides on two other boards. The first is a 16K byte static RAM board with a write-protect switch on it. The second is a 32K dynamic RAM board.

The floppy disk controller board allows limited DMA. There is 1K of RAM on that board which may be loaded without the processor's supervision. This is perhaps the only explicit hardware support of multi-tasking, since this board was designed with QUAD3 in mind, unlike the other boards.

The General

The GOLEM firmware goes by the colorful name of "The General". This firmware is resident in 4K of ROM in each GOLEM system in order to provide low-level debugging func-

tions and bootstrap support. The General will allow screen-oriented inspection and modification of memory or sectors on the floppy disk, using the cursor control features of the CRT, as well as modification of saved program registers. Real-time execution with breakpoints is optionally possible by using the software interrupt instruction.

Provision of all these features implies that the General has to do extensive interrupt processing which would potentially conflict with any operating system executing above it. This conflict is resolved as follows: When the General receives an interrupt (other than the STOP key), it searches a table in memory (one each for IRQ and NMI) for non-zero entries. These are taken to be pointers to interrupt service routines. Each routine is called in turn. When the General receives a reset, it checks for a known pattern in RAM. If the pattern is not present, it is put there and the General initializes the NMI and IRQ tables to point to various service routines inside itself. If the pattern indicates a "warm" reset, an indirect jump is taken through a special warm reset pointer - also initially set by the General.

When an operating system boots up, it should take over the warm reset pointer if desired, and replace the General's interrupt handlers with its own. This approach

keeps the amount of common knowledge between the operating system and the General to a minimum.

The General has a facility for mass loading of memory from the floppy disk. The load command will search a fixed sector on the disk for a supplied name. If the name is found, the entry will contain the starting and stopping sectors on the disk of a memory image, and the address to load it. This is the facility used to bootstrap QUAD3, or any other operating system. The system is typically loaded into the 16K RAM, after which the user throws the write-protect switch.

The QUAD2 Operating System

The QUAD2 operating system was written primarily for use in a teaching environment. This emphasis is particularly evident in the organization of the disk I/O system where flexibility, stringent error checking, and dispersed control information sacrifice some speed. Other items such as printable ASCII object files also were oriented toward the beginning user of computers. Since many of the system routines are used as class examples, there was a strong incentive to write modular, maintainable code.

Though certain parts of QUAD2 are rather simple-minded, such as the primitive command interpreter, the majority of features on the system are reasonably sophisticated. The system keeps track of the date and time which is

available from a system call. Other system calls allow fetching the command line, parsing arbitrary expressions, setting keyboard interrupt character routines, requesting an alarm clock type interrupt, and using the process tracing facilities of the General. One particular system call is especially useful, it repetitively executes a supplied subroutine for a string of filenames which may contain wild-card type characters. This allows any program to do file "family" type operations very easily.

The I/O system calls are completely device-independent. Raw and processed I/O, sequential and random-access files, and file protection are all offered. The CRT ports, the line printer and the null device all have reserved names which may be substituted for a disk file anywhere. I/O routines that do formatted data conversion and searching are also offered.

QUAD2 Commands

The QUAD2 system uses the full 16K of protected RAM available so many of the commands are actually resident. Resident commands start with an alphabetic character, and typically only the first character is significant:

B - Set or clear program breakpoints
C - Copy one or more files to a destination
D - Display directory with file information
F - Fast directory display
G - Start execution
I - Inspect memory
K - Delete one or more files
L - Relocating loader

O - Redirect standard output (normally CRT)
P - Page one or more files on the CRT
Q - Exit to General
R - Rename a file
S - Set a command address
T - Trace a programs execution
U - Unprotect one or more files
V - Print current QUAD2 version number
W - Write protect one or more files
X - Execute command file (nestable)
Z - Clear screen

In addition to the above, the left and right angle brackets may be used in front of any command to turn command prompting off or on, respectively. Also, the plus or minus characters are used to control command line echoing. These features are primarily used in command files.

Non-resident commands are indicated by typing the octalthorpe (#) and the name of the command. This causes QUAD2 to load the command using the same memory-image lookup that the General uses. The most commonly used non-resident commands are the screen editor, the absolute assembler, the file system maintenance program, and the tools presented in Chapter 2.

QUAD2 Formats

QUAD2 files are considered streams of ASCII data unless accessed in the raw mode. The null character is present at the end of the file and is not counted in the file length maintained in the directory. If a null is encountered, the file is considered exhausted, regardless of the length given by the directory entry, unless the file is

accessed in raw mode. This feature is extremely helpful in salvaging files from a corrupted diskette and is in concert with the general goal of dispersed and duplicate control information.

The extreme importance attached to file reliability in the QUAD2 system is motivated by the single-drive limitation (an economic, not architectural, limitation). Without a fast diskette-to-diskette copy capability, file backup is tedious and tends to be overlooked in practice. The principle of dispersed control information is the avoidance of irreplacable data areas on the diskette. For example, if a diskette were to have all retrieval information for files exclusively in a fixed directory area, loss of the disk area with the directory would mean a total loss of all information on the diskette.

In the QUAD2 file structure, files are doubly-linked lists of physical sectors with no requirement for contiguous allocation. The header field in each sector also contains the directory entry number of the file and the logical sector number. While this system slows down random access of files and increases sequential access time slightly, the need for annoying "squeezes" of the disk is eliminated. In practice, the file's sectors tend to be reasonably sequential.

The free sectors are kept track of by a bit map sector called the "freelist". This sector is actually kept in

memory while files are being written or deleted, for efficiency reasons. The integrity of the freelist is verified by both logitudinal and diagonal checksums.

Any track, except the directory track (0), may be excluded from the file system so that memory images may reside on one or more of the sectors. Track 4C is also reserved for maintenance information (bad blocks, excluded tracks, etc.) and contains the memory image index used both by QUAD2 and the General.

The QUAD2 file system has proven so reliable that despite some major catastrophes that have destroyed directories etc. we have never lost an important file. File recovery is aided by the disk maintenance program, CHECKD, which also aids in routine maintenance of the diskettes. CHECKD can rebuild the freelist, trace and dump files, create memory images on reserved tracks, reorganize an entire disk, or create a new file system.

Load files are kept in ASCII format on disk. This is to permit student inspection (and occasional patching) of the object. Records start with "Sxxxx" where the x's are the beginning load address of this block, and continue with a pair of hex digits for each byte. Each hex pair is separated by blanks or carriage returns. A load file may have any number of records. The QUAD2 convention is that the first location loaded is the execution address.

QUAD2 Drawbacks

The QUAD2 drawbacks were a significant source of motivation and design objectives for QUAD3. The chief drawback was the single task nature of the system. It would have been very nice, for example, to be able to use the screen editor while simultaneously printing out a file on the line printer or assembling another program.

The device-independent I/O concepts worked out well. The interrupt-driven input from the console was nice, it would have been nicer if this had been extended to output too.

The absolute originning of load files meant that programs that overwrote each other required reloading as the user switched back and forth. This was alleviated in the common cases by convention, but was always annoying. Of more serious consequence was the lack of linking facilities. Certain large programs would have benefitted from being broken into smaller files and smaller assemblies. This would have eased the strain on the text editor and improved maintenance habits.

The lack of a macroprocessor was felt due to the rather repetitive coding sequences the 6502 architecture requires, but this was not strictly a systemic drawback, rather a lack of an implementation. The same may be said about a compiled high-level language.

CHAPTER 2

SOFTWARE TOOLS FOR DEVELOPING QUAD3

The architectural impediments to multi-tasking that the 65Ø2 presents are formidable. The special nature of the first two pages in memory is one obstacle. The lack of a 16 bit register to use for re-entrant software stack manipulation is another. The third and most serious problem is the impossibility of writing load-point insensitive code, a feature offered on the newer microprocessors. Many machines which support multi-tasking have address relocation hardware to cure the problem, but this was not available on GOLEM.

The impediments mandated that relocation information would have to be included in load files. In addition, certain programming constructs and methods would have to be enforced, to avoid data corruption in the lower pages of memory. This indicated that a new assembler and loader would have to be implemented along with a "virtual machine" standard, and a scheme for relocating machine code.

Given that a new assembler was being written, certain other desirable features were specified for implementation. These included qualified names and arbitrary arithmetic expressions in operands. The most important addition was the support of external variables and routines to allow linking of separately assembled modules.

The GOLEM Relocating Assembler and Linker

The relocating assembler enforces the protocol of the 'W' machine, the virtual 65Ø2 that each process in the multi-tasking environment runs on. The output of the assembler is suitable for linking with the GOLEM linker or loading directly, if no references are unresolved.

The 'W' Machine Registers

The 'W' machine has no zero page like the 65Ø2. Instead, 16 locations in the zero page are designated as the "registers". These locations are the only zero page references permitted to processes. It is the responsibility of the operating system to insure the integrity of the registers when control swapping is done.

The relocating assembler supports the register concept by doing away with zero-page addressing in the operand field. The assembler recognizes the register definitions and uses the zero-page addressing when possible in the generated code. The register names may be given additional names by use of the REG pseudo-op.

External Definitions

The assembler recognizes the CSECT and DSECT pseudo-ops as the beginnings of separately linkable sections of code or data. A section is terminated by an END pseudo-op. Any label in front of CSECT or DSECT is defined externally. To force an external definition in a CSECT, the

ENTRY or XSR pseudo-ops may be employed. ENTRY has no other effect while XSR also begins a qualified name space. The EXT pseudo-op will force an external definition in a DSECT. All definitions inside a section that are not forced external are local to the section.

In order to clarify the above, the operation of the assembler will be explained. The assembler starts off in what is defined as "Pass Ø". At this level, only pseudo-ops involving listing control, input redirection, or global equates (including register names) are allowed. The CSECT pseudo-op will switch the assembler into "Pass 1". The assembler will process the code and local definitions, adding the local definitions to the local symbol table, while sizing the section. When END is encountered, the assembler seeks back in the file to the CSECT directive and switches to "Pass 2". Now the section will be processed again and code will be generated for the object file. When END is encountered, the assembler will switch back to "Pass Ø" and the local symbol table is flushed. DSECTs are handled similarly, but no code is allowed, only data definitions.

While processing a CSECT in "Pass 2", if the assembler cannot resolve an operand reference, it is assumed to be an external reference. This has the disadvantage of catching many typographical errors at linkage time rather than assembly time, but it avoids the tedious redefinition

of external references in each section that refers to them. In retrospect, it would have been preferable to require declaring identifiers external to the assembly in order to catch errors earlier. This means that the relocating assembler would remember the external definitions when it flushes the local symbol table, although resolution would still be done by the linker.

Qualified Name Spaces

The relocating assembler puts an upper bound of 128 characters on a symbol name, thus it avoids the annoying short variable names that many assemblers (and even some high-level languages) impose on the programmer. This helps make assembly code more readable. Another problem that faces programmers is the constant invention of unique variable names for short branches in the program. The idea of qualified names allows using short identifiers for trivial labels, by internally prepending the "space name".

The SSR and XSR pseudo-ops introduce a qualified name space and use the label on the statement as the space name. XSR also forces an external definition of the label. Thereafter, any symbol beginning with a percent sign has the percent sign replaced (internally) with the space name and a period. For example:

```
; Subroutine to increment register pair Ø
incPØ: sbr             ; (opcodes may be in lower case)
       inc  RØ         ; Increment low byte of pair
       bne  %1         ; Skip if non-zero
       inc  R1         ; Else increment high byte
%1:    rts
       esr
```

In the example, the second label would actually be held as "incPØ.1" in the symbol table. The ESR pseudo-op ends the name space. Name spaces may be nested, e.g. the label "%cat" may actually be "dog.mouse.rat.cat". Transfers to qualified labels from outside the name space may be accomplished by writing out the real name in full (e.g. jmp cat.dog) but this is discouraged from a stylistic viewpoint.

Operands and Arithmetic Expressions

Good programming style dictates parameterization of constants whenever possible. The lack of arithmetic expressions in our previous assembler had sometimes forced dependent definitions. In other words, a constant was a direct function of a second constant. Such dependencies when not automated, increase the difficulty of maintaining code. For example: "FOO equ 4" and "BAR equ 8" are inferior to "FOO equ 4" and "BAR equ FOO*2", when such a dependency truly exists.

The relocating assembler was given the capability to handle arbitrary arithmetic expressions involving symbols and constants. The most critical concern revolved around the legality of various expressions mixing different types of symbols. In particular, symbols may be either absolute,

relative or registers. It was quickly decided that arithmetic involving registers could be limited to adding or subtracting a constant from the register number. This limitation was necessary for proper addressing recognition and "made sense" from a virtual machine point of view.

The rules for expressions in the address fields are as follows. Recall that a symbol may have either the "absolute" or "relative" attribute.

Expression	Value	Attribute
symbol	defined value	defined attribute
number (must start with Ø-9 or /)	hexadecimal value	absolute
@number	hexadecimal value	relative
(expr)	expr	same as expr
+expr	expr	same as expr
-expr	two's complement of expr	illegal for relative
~expr	one's complement of expr	illegal for relative
expr!	expr	absolute
expr^	expr/256	same as expr
expr+expr	addition of exprs	sum of attributes
expr-expr	subtraction of exprs	difference of attributes
expr1>expr2	expr1 shifted right by expr2	illegal for either expr relative
expr1<expr2	expr1 shifted left by expr2	illegal for either expr relative

expr*expr	product of the exprs	illegal for either expr relative
expr|expr	logical "or" of the exprs	illegal for either expr relative
expr&expr	logical "and" of the exprs	illegal for either expr relative

The arithmetic involving attributes used by the binary addition and subtraction operators allows legal expressions to contain intermediate results that are invalid. For example, if foo, bar and bif were all relative, then the expression "foo+bar-bif" would be a legitimate relative expression, but the intermediate expression "foo+bar" would be nonsensical. In actual fact, evaluation proceeds right to left for non-parenthetical expressions so this example would not be a problem regardless.

The ! and ^ unary suffix operators were developed for the unique problems of the 6502. Since no 16 bit register exists, address constants require notation for the most and least significant bytes of the address. When the assembler expects an 8 bit operand, the least significant byte of the expression (all expressions are carried as 16 bits) is used, and the attribute is kept. This requires the following format when loading an address constant:

```
; Subroutine to load the address of foo in pair 0
ptfoo: pha            ; save the accumulator
       lda  #foo!     ; get least sig. byte of address
       sta  R0        ; store in reg 0
       lda  #foo^     ; get most sig. byte of address
       sta  R1        ; store in reg 1
       pla            ; restore accumulator
       rts
```

Note that the "!" is necessary when loading the least significant byte of the address to force the absolute attribute. This is due to the fact that we relocate only on page boundaries, thus only the page addresses are relocated.

The Relocating Linker and External Symbols

The facilities of the relocating assembler must be supported with an equally sophisticated linkage editor. The GOLEM linker was written to support all of the assembler's arithmetic operations, so that the validity of expressions is independent of whether the identifiers are externals or not. The sole difference to the user is that invalid arithmetic expressions involving externals will be flagged at link time instead of assembly time.

The linker was given additional options to produce map files, to allow partial linking, to produce QUAD2 compatible absolute files from relocating files, and to include a standard predefinition file at link time. This last feature was included with explicit QUAD3 system call support in mind.

The standard adopted for relocatable code was chosen with the same criteria we used for the QUAD2 absolute code. This resulted in printing ASCII files that read well but do take up more disk space. As mentioned, relocation occurs on page boundaries, thus each byte in a load file is either relocatable or not, and the relocatable bytes need only have the base page number added. No starting address is needed

in a relocatable file, but the total number of pages is given with "Rxx" where xx is the number of pages. Within the file, page alignment and skipping of pages is provided by "Pxx" where xx is the number of pages to skip.

Object output from the relocating assembler may have numerous sections, each beginning with an "Rxx" or "Gxx" where the latter form indicates a data section. The linker gathers the data sections and links them on the end of the load module. Tokens in the object file starting with "#" or "%" indicate a one or two byte external expression, respectively, that the linker must resolve. Tokens starting with with "*" or "=" indicate relative or absolute external definitions, respectively, which the linker adds to the symbol table.

MACP - The GOLEM Macroprocessor

The architectural features of the 'W' machine are motivated by necessity and are directly supported by the assembler. The code produced is capable of being executed under QUAD3. It is readily apparent though, that many instruction sequences are repetitive in nature. In particular, instruction sequences involving data transfer to and from the registers (which are often used in pairs, for addresses) are extremely repetitive. Such cases suggest the use of macro-instructions.

A macroprocessor improves readability of the source code, cuts down on typographical errors, and allows more

productive code generation. We thus set out to design a suitable macroprocessor.

The following features were considered essential: Conditional processing would include string matching, checking for being defined or undefined, nesting, and "ELSE" clauses. The ability to have local and global symbolic parameters, for macro recursion was needed, as was a defaulting facility for parameters. Nestable input redirection was required. String expressions should allow taking substrings. Symbolic parameters could be introduced anywhere, and taken as constants, macroprocessor commands, other symbolic parameters, etc. Simple string arithmetic and iterative processing should also be available.

The result of our development was "MACP", a general purpose macroprocessor (Williams 1982). The temptation to put 65Ø2 or 'W' machine specific features into MACP was successfully resisted. As a result, MACP may be used to generate any sort of assembly text file. The only compromise made to target syntax is the assumptions MACP makes about all assemblers. Specifically, when in "text mode", MACP is conscious of input "lines". If the first string on a line is terminated with a colon, it is taken to be a label and the next string is the "opcode". Otherwise the "opcode" is the first string on the line. In order to avoid putting assembler specific information in the macroprocessor, macro invocations are recognized in upper case, while lower case

"op code" strings (only the first character is significant) are passed through. An unquoted semicolon on a line is taken as the beginning of the comment string. This is significant in the case where parameters are purposely left off the end of a macro invocation, but a comment is desired. The output line formatting assumes that the assembler requires labels to begin in the first column of a line and to be terminated with a colon, and that all other operands may be separated by an arbitrary amount of "white space".

In "command mode", line demarcation is ignored. In this mode, strings are taken as macroprocessor control statements. A set of strings between square brackets is processed in text mode. Curly braces may be used to group strings in command mode for conditional execution. Anywhere a curly bracketed set of strings may be used, a square bracketed set may be used instead. In either mode, data surrounded by PL/I style comments (/* comment */) are ignored.

The dollar sign introduces a symbolic parameter. The characters after the dollar sign up to a period or white space constitute the name of the parameter. The value of the parameter is substituted into the input stream in place of the dollar sign and identifier. If the name was terminated by a period, it is replaced and the value is concatenated to the trailing string. Periods bind to the innermost dollar sign in nested cases. For example, if parameter

FOO had value "DOG" and parameter DOG1 had value "BAR", then "$$FOO.1" would be replaced by "BAR" in the input. Now if parameter DOG had value "ABC" then "$$FOO..1" would be replaced by "ABC1".

A string may contain blanks, but it must be quoted (with "") to be properly recognized. The special macroprocessor control characters ($, <, >, and \) may be escaped with a backslash. The left angle bracket introduces a string expression. The string expression evaluation rules are separate from the definition of MACP so that alternate string expression evaluators may be used for different processing tasks. The standard string expression evaluator allows for testing string equality and taking substrings. String expressions may be nested; MACP will handle the recursion for the evaluator automatically.

The Macroprocessor Commands

The following commands are available in command mode:

IF <expr> <group>
    If the expr is not the null string, the
    bracketed group is processed.

IF <expr> <group> ELSE <group>
    This is like the above except that
    the group following the "ELSE" is
    processed if the first group was not.

IFDEF <parm> <group>
    The group will be executed if the
    parameter in question is defined.
    An "ELSE" may also be used as above.

IFNDEF <parm> <group>
This is the opposite of IFDEF. An "ELSE" may be used as above.

SET <parm> <value>
This will define or redefine the specified local parameter with the specified value.

GSET <parm> <value>
This is like SET for global parameters.

DEFAULT <parm> <value>
This is like SET, but the parameter will be given the value only if it has not been defined before.

GDEF <parm> <value>
This is like DEFAULT for globals.

INCR <parm>
This performs a "string increment" on the specified parameter. The arithmetic is performed on trailing numeric characters in the value.

DECR <parm>
This is the opposite of INCR.

WHILE <expr> <group>
The group will be repeatedly processed while the expression is non-null. This allows repetitive text generation.

WHILEDEF <parm> <group>
The group will be repeatedly processed until the specified parameter is undefined.

INCLUDE <filename>
The specified file is inserted into the input stream at this point.

MACRO <name> <parm> <parm> ... <group>
This is a macro definition.
The name defined must start with an upper case character. The parameters represent the names of local symbolic parameters that will be set positionally at invocation time. A single dollar sign will set a parameter to null.

MACP Examples

The "MOVEA" macro moves an address to the specified location. Immediate data is supported, and this requires different treatment from a normal address.

```
MACRO MOVEA source dest {
     IF < $source:1:1 = # > /* immediate case */
          [
          lda  $source.!
          sta  $dest
          lda  $source.^
          sta  $dest.+1
          ]
     ELSE [     /* otherwise do normal case */
          lda  $source
          sta  $dest
          lda  $source.+1
          sta  $dest.+1
          ]
     }     /* end of macro */
```

The MOVEA macro is one of a set of standard 'W' machine macros that were used throughout development of QUAD3. Before the macroprocessor was available, the coding sequence that MOVEA replaced was botched regularly, introducing one more error to be removed by tedious debugging. Now that the macroprocessor is available, errors due to hastily typed repetitive sequences have virtually disapppeared. A complete listing of the 'W' machine macros appears in Appendix B.

A somewhat more complex example is provided by the PUSH macro. This macro demonstrates how a variable number of arguments may be easily handled. The PUSH macro will push up to ten values on the user stack, making use of the "PUSH" system call. Notice how no name conflict exists

between the macro and the subroutine name.

```
MACRO PUSH aØ a1 a2 a3 a4 a5 a6 a7 a8 a9
  {
     SET ctr Ø
     WHILEDEF a$ctr {
       [
         lda $a$ctr
         jsr PUSH
       ]
       INCR ctr
     }
  }
```

The limit of ten arguments to the PUSH macro was based around the tedious typing of the parameters for the macro definition. In retrospect, MACP should have contained a shorthand form for representing an arbitrary number of parameters with regular names. Such a facility could have been also used for passing the correct number of arguments to a sub-macro, based on the number of supplied arguments. For example, let "aØ*" represent "aØ a1 a2..." and "$aØ*" represent "$aØ $a1..." evaluated up to the actual number of parameters defined. If such a facility were added to MACP, a PULL macro, that would take the same arguments as the PUSH macro (in the same order) and pull them correctly (in reverse order), might be:

```
MACRO PULL aØ*
  {
    IFDEF aØ
      [
        PULL $a1*
        jsr  PULL
        sta  $aØ
      ]
  }
```

Although such a facility would have added another special

character to MACP, (special characters such as "$" must be escaped with "\" if they are to be normal text), it would have been a useful addition.

Effects of the Tools

All of the tools listed in this chapter contributed greatly to the development of QUAD3. In addition, they were all so useful that ongoing development on the older QUAD2 system used them. They proved so popular that certain features for QUAD2 were added. The relocating assembler now allows access to the entire zero page. The linker allows either code or data (or both) to be assigned to absolute addresses on output and will accept absolutly origined input too.

The relocating assembler was written by Dr. Ted Williams. The linker was written by the author. The macroprocessor has been truly a joint project. The initial implementation was written by Stan Telson, Dr. Williams, and the author. Dick Milakovich has improved it greatly.

All three programs have been, or are being, moved to work under QUAD3. The initial implementations had to run under QUAD2, of course, in order to generate QUAD3.

CHAPTER 3

OVERVIEW OF QUAD3

This chapter presents a functional overview of the QUAD3 system. Particular attention is paid here to the features implemented to satisfy the design requirement of compatibility with the precursor system, QUAD2. Later chapters will present the implementation as well as trade-offs involved in the design.

The basic QUAD3 operating system has no explicit I/O device support nor does it provide explicit terminal command support. This might be contrasted with UNIX (Ritchie and Thompson 1974) where every process has a "controlling terminal". QUAD3's approach means that it would work well in a variety of environments. For example, QUAD3 makes an excellent operating system for dedicated controller applications, while UNIX does not. On the other hand, certain features found in terminal-oriented systems are difficult to implement for QUAD3.

Since one of the design goals was to provide QUAD2 compatibility, the particular device drivers and programs written to run in this implementation of QUAD3 were a terminal driver, a disk driver, a line printer driver, a QUAD2-compatible file system, and a command interpreter. Although the thesis work comprises both QUAD3 and this

application (as well as the development tools), only the "init" portion of QUAD3 must be changed to provide a totally different environment.

I/O Support

Although pure QUAD3 does not provide support for explicit devices - relying on additional code to customize it for the application, strong unified I/O support is provided through the "extended I/O system", abbreviated as "EIO". This unified I/O system allows devices and files to be used interchangeably by the user. It also masks device characteristics (such as block orientation vs. character orientation) from the user and provides automatic buffering where necessary.

Adding Devices

EIO may be thought of as managing the entire name space of files and devices. When an "open" request is passed to EIO, the filename includes both a device specifier and a file specifier, either of which may be null. When the file specifier is null, referral is made to a device itself, as opposed to a file system residing on that device. When the device specifier is null, it just refers to whatever device has been declared to the system with the null string for its device name.

Each device, when declared to EIO, specifies the portion of the total file and device name space that is sup-

ported by that device. If the device supports a file system, the name space specifier must include wild cards such as "*" or "?" to indicate the range of acceptable file names. Devices that do not support file systems will just declare the name of the device (e.g. "lp:" is the name of the line printer).

The asterisk matches any string of alphanumeric characters while the question mark matches precisely one. "Alphanumeric" characters, in this context, are defined to be alphabetic, numeric, or the "_", ".", and "`" characters. Device names must contain at least one non-alphanumeric character; an all alphanumeric string would select the device with the null string for a name. Such a device should support a file system and is referred to as the "default" device. In the implementation for this thesis, the QUAD2 compatible file system on the floppy disk is declared as the default device.

In addition to the logical device support provided by EIO, there are several low-level system calls devoted to support of physical device drivers, including an orderly way to install or remove interrupt handlers. Interrupt handlers have partial access to the synchronization primitives described later, in order to wake up driver software after external events. The kernel supports process rescheduling in such cases.

Physical and Logical Devices in this Implementation

Physical device drivers for the floppy disk, for any number of serial I/O ports, and for the line printer have been implemented. They appear as a number of subroutines (e.g. read a disk sector). Although these subroutines are not available to the user programs, they are the base used by the logical device drivers.

The line printer is inserted in the file name space as "lp:". The driver prepends page headers to output. A null device named ":" is also implemented. It gives an immediate end-of-file when read, and swallows characters when written to. The terminals are inserted into the name space as "=n" where n is the number of the serial I/O port, starting from zero. This is actually done by the command interpreter presented in the next section.

As mentioned before, the QUAD2 compatible file system on the floppy disk is implemented with a driver. This would not preclude inserting a non-compatible disk into the floppy disk drive and using a different driver. For example, in our single-drive system, one might read a file into an editor's buffer using a FLEX (a commercially available operating system) disk in the drive and a device driver for the FLEX file system. The disk could be changed to a QUAD2 disk, and the file written out using the normal QUAD2 compatible driver. Note that the FLEX driver could make use of the physical I/O subroutines for the floppy disk.

LIZ - A Command Interpreter

The 'LIZ' (short for "Lizard Monitor") program was written to provide a high-level terminal interface to QUAD3. It depends on a low-level serial I/O driver (described later). It adds another layer of human-oriented control to the terminal and inserts the terminal into the QUAD3 I/O system, so that it may be treated like any file. LIZ is also a general-purpose command interpreter, allowing processes to be created and killed by user command.

The Virtual Terminal Facility

When LIZ is first started, it spawns a separate process called the VTF (Virtual Terminal Facility) to control the terminal I/O. The VTF declares the terminal to the QUAD3 I/O system so that it may be opened like a file. The low-level serial I/O driver does no character interpretation, so it is the VTF that provides the normal human-oriented character processing. In particular, control characters may be used to delete a previous character, delete the whole input line, pause or resume terminal output, and send a keyboard interrupt. The user may also force the end-of-file condition, with a control character, a useful feature when using terminal input to substitute for a disk file.

Any program that has opened the terminal for reading will not receive characters until the line has been finished (a carriage return was typed). This allows the user to

correct mistakes in typing. Programs that write to the terminal need not worry about device dependencies, such as appending line feeds after carriage returns.

Certain programs, such as a screen editor, need to be able to interact directly with the terminal. They can open the terminal in update mode - which changes the way the VTF handles it. Echoing and special character processing is now turned off, and characters are passed along as soon as they are received.

Command Execution

LIZ prompts the user with a dash. The user then types a line of input which LIZ takes as the command string. Blanks and tabs serve as separators, so the command string may be viewed as a series of tokens. The first token is taken as the filename of a program to load and execute. The other tokens represent arguments to the program. Certain special characters are special to LIZ and serve to break up the command line.

The ampersand and vertical bar ("&" and "|") are recognized as special characters. They separate commands, thus allowing several commands to be executed with one command line. A process is created for each command, and LIZ will wait until all of the processes run to termination, before prompting again. This is the usual mode of operation for any command that would interact with the user at the terminal.

I/O Redirection

Many programs designed to be commands under LIZ, make use of certain parameters passed to them when they are initiated. Not only are the command line arguments made available, but three files will be already opened for the program. These files are called the standard input, standard output, and standard error output - just as in UNIX.

The standard files will all be the terminal that LIZ is controlling, unless the user employs special characters on the commmand line to redirect the input or output. The greater-than, less-than, and hat (">", "<", and "^") are special and specify redirecting the standard output, input, or error output, respectively. The special character must be followed by the filename to direct the I/O to (or from).

In the case of standard input, the left brace, "{", is a special character that may replace the input filename. The brace introduces an "in-line" file. All text following the brace up to the matching right brace (they may be nested) is buffered up and appears as input to the program. The in-line file disappears when the program terminates. It may contain carriage returns; LIZ prompts with "->" to remind the user that he is entering an in-line file.

The vertical bar differs from the ampersand in that it not only separates commands, but indicates that the standard output of the command on the left is to be connected to the standard input of the command on the right. Such a con-

struct might be used to take the output of a macroprocessor, direct it to the input of an assembler, and direct the assembler's output to the input of a linker. It should be noted that the assembler and linker would have to be "single-pass", or they might write a temporary file for the second pass to read.

```
-tr A " " < {ABACADABRA
->} | wc -lw
4
1
-
```

The above example illustrates a number of the features of LIZ. The "tr" program (Kernighan and Plauger 1980) copies from its standard input to its standard output and is used for character transliteration. In this example, the two arguments specify that capital A's are to be replaced with spaces. Notice that the second argument must be surrounded with quotes so that LIZ can recognize it from ordinary white space. The quotes are stripped when LIZ parses the line. The input to "tr" is a single line immediate file consisting of the word "ABACADABRA". The output is directed to a second command, "wc" (Kernighan and Plauger 1980). This command counts words, characters and lines, although in this example, its argument specifies word and line counts only. The next two lines are the results from "wc", and only then does LIZ prompt for another command.

Background Execution and Built-In Commands

The asterisk and the question mark ("*" and "?") are considered special when prepended to the command line. The asterisk specifies that all commands on the line are to be executed in the "background" - in other words, LIZ will initiate a process for each command, and then prompt immediately for another command line. The process IDs of each command are displayed for possible status inquiries later. The question mark indicates that each command is to be executed in trace mode. A process is created for each command, but it is halted after the first instruction is executed. This allows the user to invoke a debugger program to control the execution of the command(s) under test.

LIZ has several "built-in" commands to ease process management. A built-in command is invoked as if it were a program name, but LIZ recognizes the specific name as an immediate function. The "lj" command lists the status of all previously created background processes still running. The "kill" command allows the user to forcibly terminate one of those processes. The "lm" command displays all of the load modules currently in memory.

The "lm" command lists all of the load modules currently in memory along with their addresses, and the number of processes running for each. "lm" is usually used to find where a process under trace is loaded. The "debug" command may then be used to display memory and to single-

step the process or set breakpoints.

The other built-in command is the "liz" command - which instructs LIZ to spin off a copy of itself. This is typically used with redirected input, in order to provide a "command file" facility. Options allow echoing prompts and/or input to the terminal.

The QUAD3 Program Environment

In order for programs to work in the QUAD3 environment, certain restrictions must be placed upon them. The tools presented in the previous chapter take care of generating most of the information needed. For example, the relocating assembler generates needed relocation information, and limits access to the 65Ø2's direct page to the W machine registers.

Specific instructions for programming in the QUAD3 environment, along with complete descriptions of the system calls, are available in the QUAD3 Programmer's Manual. Instructions for writing logical and physical device drivers are also included. This manual has been reproduced in its entirety in Appendix A.

Load Modules

All load modules must be relocatable - the output of the relocating assembler or the linker. They are required to have a 24 byte area in the front which is used by memory management. The contents of the 24 byte area when the load

module is loaded is irrelevant, except for flags in the 24th byte. These flags may be used to indicate to QUAD3 whether the program is reusable and whether it is re-entrant. The first instruction of the program must follow the header immediately.

A program is reusable if it does not depend on the load process to initialize its data area. Normally, a program is re-entrant if it uses only dynamically assigned memory for data storage, although a re-entrant program could be made with a static data area using a process-unique index to only access certain parts of the static area. Specification of reusablility and re-entrancy helps QUAD3 avoid reloading a program when an existing copy is already in memory.

System Calls

A rich assortment of system calls are available to programs running under QUAD3. These are accessed by calling an external subroutine name. The linker uses a file of definitions to resolve these references.

Extended I/O System Calls. The EIO system presents a unified I/O interface to the user program. Error codes are standardized across the system. Buffering is automatically provided, so that any convenient amount of data may be input or output with one system call.

Files or devices are opened by name with the OPEN call. An eight bit number called the file descriptor is

returned. This number is used in all future correspondence regarding the open file. Arbitrary amounts of data may be read with the READ or READL calls. The latter will stop on a carriage return. Arbitrary amounts of data may be written with the WRITE or WRITEL calls, where the latter uses a null data byte rather than a count to end the write. For convenience, the READC and WRITEC calls read or write a single byte.

The SEEK call allows changing the location in the file where the next read or write will begin. The STATUS call returns the current location, and the size of the file. An explicit indication is also given of whether the file is considered in the "end-of-file" state. In the case of serial communication lines, this indicates whether a character is available for immediate input or whether a read would suspend the reading process. Such a construct is necessary for making network driver programs. Many operating systems are deficient in this respect.

The RENAME call allows changing file names or deleting files. If this is tried with a device, an error results. The GETHEX and PUTHEX calls are provided for convenience as the operation of inputting and outputting hexadecimal numbers is a frequent one in the microcomputer world. Logical devices are declared to EIO with the CONNECT system call and removed from the hierarchy with DISCONNECT.

The USER Stack. The 6502 forces the program stack to be in the second page of memory. This stack, hereafter referred to as the "hardware stack", is used by interrupts and the "Jump to Subroutine" instruction. User programs may also use it to save data. Since QUAD3 is a multi-tasking system, use of this resource must be multiplexed. QUAD3 programs are restricted to using about one fourth of the hardware stack page. This is to limit the amount of time it takes to switch processes.

To compensate for the small hardware stack, processes are given a "software stack", a page of memory reserved for the process. Programs can easily expand their user stack past one page if need be. The highest register pair, P14, also called the user stack pointer, is used to manage the software stack. P14 is correctly initialized when the process is started. A variety of system calls are available to push and pull registers to and from the user stack.

System Calls for Process Control. The SPAWN system call is the manner in which new processes are brought into being. The parent process supplies the starting execution address of the child. The parent is given back the process id of the child, while the child receives the process id of the parent. These process ids are used in other system calls.

The SUSPEND system call is used by a process to stop executing. It will remain in limbo until another process uses the RESUME system call to reawaken the suspended process.

A process may terminate itself with the EXIT call. It may also terminate another process with the KILL system call. EXIT allows the process to supply a final status. A process may "wait" on one or more processes with the WAIT system call. A list of processes is supplied, and the caller is suspended until one of the processes hits a breakpoint, suspends itself, or dies. In the last case, the final status of the dead process is made available to the waiter.

Two fundamental synchronization primitives P and V are supplied. They manage a semaphore facility. These are most often used by device drivers, but any user program may designate a byte as a semaphore byte and employ the primitives.

Memory Management System Calls. Each process executes in a certain code body in memory. A code body may be thought of as the result of a load module being loaded (with necessary relocation) into memory. These code bodies are labeled, and records of how many processes are currently executing in the code body are maintained. Thus, when a request to load a module is made, a check is first performed to see if the same module has been loaded into memory

already. If the code body in memory has no processes currently executing in it and it is marked reusable, the load need not be done. Even if it has processes executing in it, if it is marked re-entrant it can be reused.

The BECOME system call is the manner in which a process switches the code body that it is attached to. A filename is supplied, and if the necessary code body cannot be found, the file will be opened and the load module loaded. The old code body will have one less process and the new code body will have one more. The caller will only regain control if the system call fails due to a non-existent file, a non-loadable file, or lack of memory. BECOME has an option where the process stops for tracing after executing the first instruction of the new code body.

The ALLOC system call is used to obtain contiguous free memory in increments of one page. ALLOC will discard unused code bodies in memory if that is necessary to get the amount of storage requested. The FREE system call returns memory. When a process exits or is killed, all of its files are closed and its memory returned, so FREE is needed only if the process wishes to return resources and keep executing.

Programs that Work Under LIZ

In addition to the standard QUAD3 features and restrictions, certain additional considerations may apply to programs designed to work well under LIZ. Although no

program is required to use the passed parameters, effective synthesis of multi-program streams would not be possible with programs that ignore the standard input and output file descriptors passed to them. If a program takes its input from the passed input file descriptor, and sends its output to the passed output file descriptor it may be used as a "filter" (Ritchie and Thompson 1974).

LIZ also supports programs that must access the terminal more directly by passing the file name of the terminal. Those programs can then open the terminal in "update" mode - which instructs the VTF to pass unprocessed characters in both directions.

Programs started by LIZ get a register initialized to the number of arguments to the command. This count will always be positive because the command name is treated as the first argument. A register pair points to the parsed command line. Arguments may be easily retrieved with the GETARG external subroutine (actually a part of LIZ).

CHAPTER 4

THE KERNEL

This chapter discusses the implementation of the QUAD3 kernel. Attention was paid toward identifying the minimal set of functions that must have the kernel attribute, both for efficiency and stylistic reasons. This sets QUAD3 apart from numerous operating systems.

Many other systems do not attempt to minimize the part of the operating system that they consider the "kernel" (Bayer and Lycklama 1978). In some implementations, there are what is called "kernel processes", which represent the image of user processes getting kernel services (Thompson 1978). Such "kernels" often have I/O drivers, memory allocation routines, etc. that go far beyond the QUAD3 philosophy of the minimal kernel.

A Minimal Kernel

The salient characteristic of code labeled "kernel" code, is that it must run in an uninterruptable fashion. It must, of necessity, contain the code which controls the most basic resource of the system, the CPU. In other words, the process scheduler must be part of the kernel. As a very minimum, it must receive control whenever a process wishes to cease running. It should also receive control when a hardware exception occurs, indicating that a process has

gone "haywire", providing such hardware capability exists. If the system uses "time-slicing", i.e. schedules based on intervals of time, the scheduler must receive control on a regular basis with an interrupt from a hardware timer.

It also follows that the kernel must contain mechanisms for creating and deleting processes, unless the system has only fixed, immortal processes. Although much of the bookkeeping for process creation and termination may be done outside the kernel, there must be an orderly way to make the scheduler function aware of such a state change, in order for a new process to be scheduled or to prevent a defunct process from being scheduled.

Ignoring interrupts for the moment, we divide the processor use into two classes: either the scheduler is running or a process is running. In the former case, the scheduler will decide directly which process to run, initialize the machine state correctly (referred to as a "context-switch") and begin executing the proper code - thus reverting to the latter case. As a process executes, it may decide to terminate itself, or at least to suspend execution for a while. Thus, again, a mechanism must exist for a process to signal the scheduler, in order to relinquish control.

The general mechanism for signaling the scheduler is referred to as a "kernel call", also known as system calls or traps in other systems. The kernel call represents uni-

lateral surrender of control by a process, typically with a request for a certain action. We have already identified three such actions: create a process, terminate a process, and simply reschedule.

If process synchronization is needed, additional kernel calls must exist. Any mechanism that provides for suspending a process until the occurrence of some event, involves the scheduler marking the process blocked, and then later marking the process ready again when the event occurs. It then follows that synchronization primitives, if they exist, must exist in the kernel.

Interrupt handling (other than the interval timer and hardware exception interrupts) need not necessarily occur in kernel code. If, however, a process is blocked upon an event later signaled by a processor interrupt, then there must be a way for the interrupt handler to signal the kernel that the event has occurred.

Process Scheduling, Life, and Death

The QUAD3 kernel does support dynamic creation (up to some maximum) and termination of processes. The QUAD3 kernel runs with both maskable and non-maskable interrupts disabled (the latter due to off-chip hardware) so that the code may not be interrupted. Kernel calls result in a simulated interrupt, followed by a vectoring to the particular kernel function. When the function is complete, the

scheduling portion is then begun. Timer interrupts just result in direct vectoring to the scheduling function.

The scheduler works on a round-robin basis considering priority. When a scheduling is done, the highest priority process is selected. If that process is currently running, and there is one or more of equal priority ready to run, the next one is selected. Process data is kept in a table (the 65Ø2 architecture favors such an organization, as opposed to a linked list) and a scheduling requires one circular search of the table.

This scheme is very fast, and has worked well in practice. There is, however, a theoretical deficiency. Consider a high priority process in table slot 5 and two other processes with equal but lower priority in table slots 6 and 7. Assume further that processes 6 and 7 do no I/O hence they will always use up their time "slice" when scheduled. Whenever process 5 ceases to run, process 6 is selected. If process 5 normally becomes ready again before process 6 is time-sliced out, process 7 will be starved for time. In order to overcome this type of problem, it would be necessary keep track of the last process to run for each priority level.

The process table used by the kernel contains critical information about the processes. It resides in the zero page in memory, and is often read by system code not in the kernel. It may even be written to by such code, provided it

is done in a race-free manner. In order to create a new process, all the kernel must do is make another entry in the table. For simplicity, when the kernel is called to create a new process, it just makes a duplicate copy of the table entry for the calling process. The kernel call is referred to as a "cloning". The calling process may distinguish itself from the child process by referring to the process number in low memory. The child process might then do other things to set up his operating environment; the key concept is that this happens during process time, without kernel help. This keeps the kernel small and efficient. Process termination is also easy, the kernel marks the table entry as "dead" and reschedules.

Fundamental Process Synchronization

Process synchronization is needed in all but the most trivial of multi-tasking systems. As long as any resource exists which is not dedicated to a single process, there is a need to prevent two or more processes from attempting to use the resource at the same time. Such a resource might be as simple an item as additional allocated memory. At any rate, there will be certain variables in memory that control the disposition of a resource under contention. These variables must be accessed in a systematic way, free from concern about simultaneous access. It follows that a synchronization mechanism must either manipulate these variables directly, or "lock" them so that one process

alone (Hoare's (1974) "monitors" would be an appropriate example), may access them until it "unlocks" them.

For QUAD3, Dijkstra's (1968) "P" and "V" semaphore operators were chosen. Any byte in memory may be viewed as a semaphore. A P operation on a chosen semaphore will increment the semaphore if it is less than or equal to zero. If it is already one, the process that did the P will be blocked until the semaphore goes to zero. At that time, the process is reawakened and the semaphore goes back to one. A V operation on a chosen semaphore decrements it. If the decrement takes the semaphore to Ø, a search is made to see if any processes are blocked, trying to complete a P on the semaphore. The highest priority process (if any) is then allowed to complete the P.

A simple locking function to protect critical variables can be implemented with P and V. A byte is initialized to Ø and designated as the lock variable. When a process wishes to access the critical variables, a P is done on the lock variable first. After completing the access, a V is done on the lock variable. This construct is used repeatedly throughout the system code.

P and V may also be used to control multiple allocation of a homogeneous resource. For example, when accessing the floppy disk, the low level driver code must allocate 1 of 4 special RAM pages where DMA may take place. A semaphore, initialized to -3, controls the allocation. When a

process wishes to acquire one of the 4 buffers, it does a P on the semaphore, in order to guarantee that a buffer is available. The first four processes that did the P would continue on through to get a buffer, but the fifth process would be blocked until a process released a buffer and did the V.

This approach elegantly solves the problem of making processes wait for a resource. Note however, that additional synchronization is needed once it is identified that some buffer is available, in order to determine which buffer to allocate without a hazard. Here again, a semaphore could function as a lock.

The need to lock a resource for a short time occurred rather frequently in the system code. When the lock time was a matter of a few instructions, the overhead of setting up and maintaining a semaphore was significant. For this reason, a feature analogous to disabling all interrupts was deemed desirable for the system code alone. The "DIVE" and "SURFACE" subroutines allow a process to prevent rescheduling while accessing a critical data area. DIVE just increments a special "submerged" flag in the zero page. SURFACE decrements it, thus calls to the two routines may be nested. The scheduler pays homage to the flag by avoiding context-switching while the flag is non-zero. If a time-slice runs out while the "submerged" flag is set, the scheduler sets a second flag which the SURFACE routines

recognizes. When the flag is set, SURFACE will immediately call the kernel after decrementing the "submerged" flag.

It should be noted that the DIVE and SURFACE routines are not kernel calls per se, nor are they general purpose lock routines, since they prevent all other processes from running - rather than those processes that would conflict over a critical area. Thus they are not available to the user, but are used by low-level system code to protect small portions of otherwise race-sensitive code (typically a dozen instructions or fewer), primarily because they are very efficient.

I/O and Other Interrupts

The QUAD3 kernel does handle all interrupts initially. This is brought about by the lack of vectored interrupts on the 6502. The maskable interrupt line is available for any user device, but all interrupts vector initially to the kernel. Since the source of the interrupt is unknown, the kernel calls various interrupt handlers in sequence, until one of them acknowledges processing the interrupt.

These interrupt handlers must have been previously declared to the kernel, necessitating the final two kernel calls: "INSTALL" to declare such a handler, and "RIPOFF" to remove it. When an interrupt handler indicates that it has processed an interrupt, the kernel does not poll the others.

The handler has the additional option of specifying a V operation on any semaphore, when it returns to the kernel.

The kernel does not normally reschedule on interrupts, but if the handler requests a V operation, and a process becomes unblocked, a rescheduling is done. The initial implementation did reschedule on all interrupts, but performance was significantly degraded.

The non-maskable interrupts are reserved for the system. One of them is the interval timer. This timer is restarted whenever a process is scheduled in. It provides a maximum time to run, so that more than one non I/O-bound task may coexist and receive processor time. The other non-maskable interrupt is the instruction step facility. This is an interrupt that occurs on each instruction of a process being debugged. The kernel marks the process as "stopped for trace", and does a V on a special "death semaphore".

The "death semaphore" is the kernel's only other consideration to the outside world, besides the "submerged" flag. A V is also done on this semaphore when a process dies. A daemon process known as the sexton does P's on the semaphore. The sexton is presented in chapter 5.

The only other "interrupts" are from the software interrupt instruction. This is treated as though the executing process had had a trace interrupt. This allows a

debugger to use breakpoints, rather than stopping a process on each instruction.

The system reset causes the kernel to initialize the process table, create a single process and schedule it. This process, called the "init" process, is then responsible for bringing up the rest of the system.

CHAPTER 5

PROCESS CONTROL

Having presented the QUAD3 kernel, the next three chapters focus on the remaining operating system code. Most of the rest of QUAD3 is code for "system calls". This term is used herein to denote a set of subroutines available to user processes, which provide support for kernel calls, control I/O, allocate memory, etc. The system call represents actions taken under the auspices of a process, rather than surrender of control to the kernel.

Those system calls that allocate system resources (such as memory or I/O peripherals) are "monitors" in the classic sense (Hoare 1974). They have all of the code and data structures to manage the resource, and provision is made so that the code can be entered only by one process at a time. This provision might vary from call to call however, with either semaphores or DIVE usually employed.

Such an approach differs sharply from the "manager" approach (Jammel and Stiegler 1977). A manager is a separate process created to control a critical resource. Access to a resource is obtained by sending a "message" (the usual form of inter-process communication in such systems) and waiting for a message back with permission. While the manager concept is elegant, it involves several context

switches and has been proved to be less efficient than the monitor approach in a single processor system (Crowley 1980).

Other systems do not draw the differentiation between what QUAD3 calls "system calls" and "kernel calls", usually because they are the same. For example, in the UNIX operating system, all system calls are accomplished by a trap instruction which switches the processor to "kernel mode" (Thompson 1978). This means that the I/O drivers etc. operate in the machine's privileged mode. This would be contrary to the philosophy of QUAD3, where the kernel has been minimized.

The only other system code, besides the kernel and system calls, is the one "daemon" (from Greek mythology, meaning "friendly spirit") process responsible for cleaning up after dead processes. This process is known as the "Sexton".

System Calls for Kernel Interfacing

A user program does not do kernel calls directly, nor would it read data from the zero page where system information is kept. These addresses etc. are not available to the user program. What is available, as mentioned in chapter 3, is a series of external names for the available system calls, that the linker will resolve. Thus all user program interaction with the operating system is via a set of subroutine calls.

The P and V semaphore functions are available to user programs via system calls. The system call just executes the associated kernel call. Most user programs though, use the more advanced synchronization functions described later.

Process creation is provided by the SPAWN system call. The calling program provides the address where the child process should start executing. SPAWN notes the process number by reading it from low memory and then uses the CLONE kernel call. The code right after the cloning can determine if it is the parent or the child process by comparing the process number of the parent with the one in low memory. The parent code just returns with the child's process ID in P0 (register pair 0). The child code allocates a user stack of one page, and then transfers to the spawn address.

Process termination is handled by the EXIT system call. EXIT makes sure all of a process's files are closed by calling an internal EIO routine called CLOSALL, presented later in the text. The user program is allowed to give a 16 bit "exit code". EXIT places that code in the process table entry for the process. The DIE kernel call is then employed.

As mentioned, the process table is in low memory where certain system calls may modify it, so long as they do it in a race-free manner. The system code often employs

DIVE and SURFACE to avoid races). The internal routines APL, RPL, and SPL are good examples. They advance, retard, or set a process's priority level by directly writing into the process table entry for priority. APL also saves the old priority on the user stack so that RPL can restore it. These two calls are primarily for I/O system code, so that a process's priority may be moved up while it uses system resources like the disk.

The KILL system call interacts with the above three system calls. KILL works by changing the saved program counter of the victim to point to the EXIT system call, along with a status of "KILLED". What KILL may not do, is kill a process past a fixed priority level reserved for those owning I/O resources. If the owner of the floppy disk were killed, the disk would never become available. To prevent this, if the process is above the I/O priority threshold, a mark is made, rather than a direct kill. When RPL is called at the conclusion of I/O, the kill is completed. The only drawback to this scheme occurs in the case of a "stuck" I/O device - in which case the process would be unkillable.

Other Synchronization Constructs

The P and V synchronization constructs provided by the kernel were quite effective when writing device drivers and I/O interrupt handlers. They also proved useful in other areas. What they lack, however, is a way for a

process to suspend itself pending occurrence of any one of multiple events. Once a process has performed a P operation on a semaphore and become blocked as a result, only the event signified by the V operation on that semaphore will wake it. This represents an obstacle in the situation where a process would like to suspend pending the occurrence of any one of a set of possible events. A salient example is the case of a console monitor program, such as LIZ, that allows creation of several processes and is interested in the eventual disposition of each one.

This particular case was crucial to the system design, and forced creation of the Sexton process, as well as a reserved semaphore, (called the "death semaphore") that the Sexton would perform a P operation on. Whenever something happens that significantly changes the state of a process, a V operation is done on the death semaphore. This impacts the kernel in the least way possible, as it must do the V only when a process dies or hits a trace breakpoint (hardware or software).

A "V" on the death semaphore may be viewed as a logical "OR" of what would otherwise be various V operations on other semaphores. Initial experiments had dedicated semaphores for each process, which had V's done when the process died. The problem of waiting for more than one process at a time made this unworkable.

The WAIT System Call

When a process calls WAIT, it supplies a list of process IDs in a tabular form. The address of the table is the actual passed parameter. The WAIT system call code DIVEs and searches the process table to ensure that all entries represent existing processes. This is necessary to prevent a race condition where a process had already died and been processed by the Sexton (buried?) before the WAIT call had been made. If such a condition occurs, the code SURFACEs and returns to the caller with approriate indications.

If all processes in the wait table do indeed exist, the WAIT call places the address of the table in the semaphore address slot of the process's entry in the process table. (This slot is where the address of the semaphore resides when a process is blocked waiting to complete a P operation.) "Blocked" status is then placed in the status entry, SURFACE is called, and the "schedule" kernel call is done - thus blocking the calling process. The Sexton is responsible for unblocking the calling process when one of the listed processes changes state.

The SUSPEND and RESUME System Calls

The death semaphore concept allowed waking up the Sexton daemon process on either the death of a process or when the process was stopped for tracing. It proved desirable to add a feature where a process could block itself in

a less destructive fashion than suicide, yet have the state change communicated to any waiting processes. A new state, "suspended", was added as a possible process state and two new system calls were also added.

The SUSPEND system call DIVEs, places suspended status in the process table status slot, does a V on the death sempahore, and then SURFACEs and does a "schedule" kernel call. This approach required no additions to the minimal kernel.

The RESUME system call is intended for a waiting process who discovers that a process of interest has suspended. RESUME will restart the process by simply marking the appropriate entry in the process table as "ready". SUSPEND and RESUME are ideal for the synchronization mechanism for coroutines. They do not exhibit the race problem of UNIX's event/wait mechanism (Thompson 1978) that forces the operating system to play "games" with the process priorities to avoid a deadlock.

The Sexton

The immediate advantage of having a daemon process to cope with other defunct processes was that the chore of memory deallocation could be pushed to this process, the Sexton. This eliminated the problem of having a process deallocate its user stack, etc. while still running.

The Sexton does a P on the death semaphore. When the P operation completes, it searches the process table for

processes that are either dead, blocked for trace, or suspended. Dead processes have their memory allocations returned (refer to the discussion of FREALL in the next chapter). The dead process's 16 bit return code is read from the process table. For any of the three cases, a second search of the process table is performed, looking for blocked processes. The wait table for each blocked process is examined, and if it has the process ID of the process which has just died, suspended, etc., the 16 bit return code is put in the table and the blocked process is marked ready to run. There are special return codes for processes that have suspended or are blocked for tracing.

The chief hazard of this approach is the possible loss of returned status from a process. This may happen in one of two ways: either the process dies before the waiting process can complete the WAIT call, or one process dies, the waiting process is unblocked, but the other process (which was also being waited on) dies before another WAIT call. In practice, it is parent processes that typically wait on child processes. Both of these hazards may be eliminated by running child processes at a lower base priority than the parent. LIZ does this. The Sexton itself runs at an extremely high priority.

CHAPTER 6

QUAD3 MEMORY MANAGEMENT

Chapter 1 explained the amount and type of random-access memory available on the GOLEM system. The QUAD3 system code resides in the 16K of static RAM starting at location 8000. This memory can be write-protected by a switch on the board once the operating system has been loaded. 4K of system RAM starts at address C000, and various ROMs, including the General occupy the range from F000-FFFF. I/O devices are mapped in the D000-DFFF range, and so is the DMA diskette controller RAM.

The 32K of dynamic RAM from 0 to 7FFF is the area for user programs. The first 6 pages are carved away for the zero page, hardware stack, and hardware stack save areas, thus available memory runs from 600 to 7FFF. Memory in this area is used for programs, user stacks, and dynamically allocated data areas. It is referred to as "user" memory.

Basic Memory Allocation

User memory is controlled by a map with one byte in the map corresponding to a page of memory. Memory is allocated in page increments. The ALLOC system call returns a requested number of consecutive pages of free memory by scanning the map for such a chunk. When ALLOC parcels out

memory, it marks the process number in the map for each page. The FREE system call allows returning one or more consecutive pages. FREE just marks those pages as free again.

ALLOC uses a "first-fit" algorithm when searching for a contiguous area of free memory large enough to fulfill a request. In practice this has worked very well. Fragmentation of free memory has not been a problem because of the relatively high percentage of allocation requests for single pages. This high percentage stems from all system code, and most user programs, getting data areas a page at a time.

QUAD3 Load Modules

AS mentioned in Chapter 3, all QUAD3 programs are required to have a 24 byte header at the beginning of the load module with the 24th byte indicating re-entrancy and reusability. Each process has a corresponding load module associated with it, as indicated by an entry in the process table pointing to the load module. System processes are an exception, they have Ø in this entry. A process may change which load module it is associated with via the BECOME system call. The header information for the load module contains: the name of the file where it was loaded from, the number of current processes associated with this load module, and pointers to keep the load modules in a doubly-linked list.

When a process does a BECOME, it supplies a file name where the desired load module resides. The BECOME code first examines the load modules already in memory to see if the named file is already loaded. If it is loaded, a check is made to see if any processes are currently associated with it. If it has other processes running, but is re-entrant or if it is merely reusable and has no processes currently running then no load need take place.

If a load has to be done, the first token of the load file (the Rxx token as explained in Chapter 2) to determine how much memory to allocate. When memory is allocated for a load module, it is not marked with a process id but rather a separate number called the "soul". The module is loaded and relocated, and the load module header is initialized with the file name, the soul, and the forward and back pointers to other load modules.

Whether or not a load had to be done, once BECOME has obtained the new module, the process count in the header is incremented and the load module pointer in the process table is updated. The old load module has its process count decremented. BECOME normally transfers to the execution address of the new load module. The exception occurs if the caller specified that execution should proceed in trace mode. In that case, the BECOME code will not directly transfer to the new load module entry point, but will employ

the General's trace facility to transfer there with a trace interrupt.

Removing Old Load Modules

When the Sexton encounters a dead process, it calls an internal memory management routine called FREEALL. This routine will free all of the memory allocated by that process. It will also inspect the header of the load module associated with the deceased process. The process count is decremented and if other processes remain, no further action is taken. If the load module now has no processes associated with it, a check is made to see if it is reusable. If it is not reusable, the load module is unlinked from the chain and the memory is freed (using the soul number).

When ALLOC cannot fulfill a request for some number of contiguous pages, a check is made for modules that have been left in memory because they are reusable, but have no processes associated with them. Such modules are removed until there is enough memory to satisfy ALLOC's needs.

CHAPTER 7

THE EXTENDED I/O SYSTEM

The Extended I/O system (EIO) represents the most innovative portion of QUAD3. Like any good unified I/O system, it allows devices and files to be used interchangeably by user programs, and it masks particular device dependencies behind a simple character-oriented set of I/O system calls. The unique part of EIO is the ability to dynamically add either real or virtual devices - apportioning part of the total file name space with them. These dynamic devices may even support file systems.

Managing Devices

A logical device is declared to the EIO system with the CONNECT system call. The P2 register points to a name string with possible wild cards (as discussed in Chapter 3) which specifies the portion of the file name space that this device wishes to respond to. The PØ register points to a table of seven addresses, which are the device handler routines for opening, closing, reading from, writing to, seeking in, obtaining status about, or renaming/deleting files on that device (or just the device itself if it supports one file). An entry may be Ø if no such function is allowed (e.g. renaming or reading from the line printer). EIO keeps both the device name string and routine table

pointers in a table called the "device table". The DISCONNECT subroutine merely removes those entries from the table.

User EIO Calls

When a process wishes to open a file, it uses the OPEN system call. It must supply the name of the file and a code for whether it intends to read, write, or update (read and write) the file. The EIO system scans over the device name strings looking for a match with the name of the file. When a match is found, EIO calls the device's open routine. The device open routine must either return an error indication, or it must return a device "index" to be used to refer to the now open file. The device open routine also indicates the block length of data transfers that it performs.

If the open was successful, EIO makes an entry in the "open file table", a system-wide table describing all open files. The process id, device pointer, device index, and blocking factor are all stored. In addition, EIO allocates a buffering area for data for that file, and saves the buffer address. The OPEN call then returns the entry number in the open file table as the "file descriptor" to the caller. The caller must then use the file descriptor in all other communication about the open file.

When a process decides to rename or delete a file, it calls the RENAME system call. PØ points to the old file name and P2 points to either the new name or to a null if the file is to be deleted. EIO scans the device name

strings in the same fashion as an open, and calls the appropriate device rename routine.

When a process wishes to read data from an open file it uses the READ system call. The process supplies a pointer to the area to read the data into and a count, as well as the file descriptor. EIO uses the file descriptor to look into the open file table and get the buffer area for that file. Data is moved from the buffer area to the destination until the count is satisfied or an end of file condition occurs. Whenever the buffer area is depleted, EIO calls the device read routine with the correct device index and the buffer address. The device read routine will place up to a full block's worth of data in the buffer.

An exception is made for device routines with a block size of one - these are called character devices. EIO does not allocate a data buffer for character devices but reads the characters one at a time and moves them straight to the destination area.

There are two variations on the READ system call, READL and READC. The former is almost identical to READ, but it terminates when the ASCII carriage return is sensed in the input - as an aid to line-oriented programs. READL will also stop when the count is reached if no carriage return has been seen. READC is a system call to get a single character from an open file.

When a user program wishes to write data to a file, the operation is analogous to READ, but the WRITE system call is used. The EIO system takes the characters from the specified area and either buffers up a blocks worth in the file's data buffer or calls the device write routine for each character if it is a character-oriented device. In the former case, each time the buffer fills up, the device write routine is called. The WRITEL routine is similar to WRITE but it will stop when a null is encountered in the data. WRITEC is the single character write.

EIO keeps track of the overall position in the file. When the user program uses the SEEK system call (for a read file) in order to change the location in the file, EIO computes whether or not it must change the contents of the data buffer. For example, if the user program had read only 5 characters from the a 200 byte data buffer, and then did a seek forward of 100 characters, EIO would not have to call the device seek routine but would simply update where it would take the next character in the buffer. If EIO has to call the device seek routine, the buffer address and the overall seek location are given. The device routine must fill the buffer with the appropriate block and then indicate to EIO where in the buffer it must start parcelling out characters.

A SEEK in a file opened for update is a little trickier. EIO keeps track of whether any of the data in the

buffer is data written by the user. If a seek requires the buffer to be changed, the old block is written (if required) with the device write routine before the device seek is done.

When the user program does a STATUS call the EIO system can return the file position since that is maintained, but it must call the device status routine to see if the file is exhausted and what the file size is.

Ancillary EIO Support

There is a standard set of EIO error codes which the device drivers are expected to adhere to when they return error statuses. The IOERR system call is a handy way to translate these error codes into English error messages.

Output of hexadecimal numbers is such a common operation that a system call named PUTHEX has been provided. PUTHEX takes a file descriptor like any EIO call but it actually uses WRITEC to do the output.

CHAPTER 8

LIZ AND THE VIRTUAL TERMINAL FACILITY

The LIZ command interpreter and Virtual Terminal Facility (VTF) represent the most visible "user" programs running under QUAD3. LIZ receives no special dispensation from the QUAD3 system other than being automatically initiated as part of system startup. Pure QUAD3 does not require LIZ, however most systems do have terminals for user input, thus a terminal driver and command interpreter were included in the system development. This chapter discusses the implementation of LIZ, the VTF, and virtual devices for LIZ called "pipes". These programs make efficacious use of the basic QUAD3 features, thus they serve as an example of the power and elegance of the QUAD3 system for specific implementations.

LIZ has a small amount of startup code. The startup code must be passed a physical port number (the number that identifies the port to the low-level serial I/O routines). The VTF, the process that converts the raw terminal I/O into processed lines, is then spawned and its process ID is saved. The startup code then initializes the registers as if LIZ had spawned itself and enters the main body of the code. The main body is re-entrant so that LIZ may indeed

spawn itself as a sub-process; this feature is used in the "liz" built-in command mentioned in Chapter 3.

VTF Implementation

The VTF process saves the process ID of its associated LIZ. It then does a CONNECT of "=n" where n is the ASCII representation of the serial I/O port number. All VTF variables are stored in a table indexed by the port number, thus the VTF code can be re-entrant and still have a static data area. The VTF code comprises both the VTF process code and the virtual terminal device code, which was the object of the CONNECT.

VTF Process

The VTF process advances its priority to run at a device-level priority, in order to ensure fast character service. Several flags and semaphores are initialized. One of the flags is the "raw mode" flag (initially false). When the raw mode flag is true, all input characters are treated as data.

The VTF uses two semaphores to control task activation for input. One semaphore is called the "raw" semaphore. When the raw mode flag is set, a process reading characters (via the VT device read) will do a P on this semaphore, enabling the VTF to wake him up on each character. If raw mode is not in effect (the usual case), the process will do a P on the "line" semaphore, which prevents

him from being awakened until the VTF sees a carriage return or a logical "end-of-file". This is elaborated on in the following paragraphs.

After initializations, the VTF now loops - accumulating characters by calling the serial I/O driver. When a character is obtained: If raw mode is set, the character is placed into a circular input buffer and the raw mode character semaphore has a V operation performed. If raw mode is not set, the character is examined to see if it is a special character. If it is not, it is just placed in the buffer, the line length is incremented, and the character is echoed back with the serial I/O output routine. If it is a carriage return or ^D (CONTROL D), it is placed in the buffer, a carriage return and line feed are echoed, the line length is set to Ø, and the line semaphore has a V operation done on it.

The ^E and DELETE characters are special characters. If the line length is Ø, they are ignored, otherwise a backspace, space, backspace sequence is echoed, the line length is decremented, and the circular buffer has the last inserted character plucked back out. The ^U character (delete line) is also special. The same action is taken as for ^E, except it is repeated until the line length becomes Ø.

The ^S (pause display) and ^Q (resume display) characters are also special. They employ a semaphore called the

hold semaphore that is initially Ø, and a flag called the hold flag that is initially false. When a ^S is detected, if the hold flag is already set, the key is ignored. Otherwise, the hold flag is set and a P is done on the hold semaphore. When a ^Q is seen, if the hold flag is false, it is ignored. Otherwise, the hold flag is cleared and a V is done on the hold semaphore. The VT device write routine will be responsible for doing a P and V on this semaphore as discussed later.

The ^C (keyboard interrupt) key also requires special treatment. The VTF inspects the process table to see if LIZ is currently blocked (i.e. running sub-processes). No race can occur here because the VTF runs at a higher priority. If LIZ is not blocked, then ^C is the equivalent of a line delete. If LIZ is blocked, then the VTF will do a SUSPEND, which will result in the Sexton waking LIZ. LIZ is responsible for RESUMEing the VTF.

The Virtual Terminal Device Routines

When EIO calls the VT OPEN routine, it will pass the READ/WRITE/UPDATE flag on. The characteristics of the virtual terminal vary greatly depending upon that flag. Many different processes can have a given virtual terminal open at the same time, though each instance is considered a separate virtual file. If several processes attempt to do I/O at the same time, reasonable interpretations are provided.

When a process opens a virtual terminal, an entry is created in a table used for all virtual terminal opens. This entry includes the number of the port, the mode of access, and a bit indicating whether or not the "end-of-file" condition has occurred. The device index returned to EIO is the entry number in this table.

When a process opens a virtual terminal for reading, EIO is told that the transfers will be in blocks of 128 bytes - the maximum input line size. When a process opens a virtual terminal for writing, EIO is told that it is a character-oriented device. When a process opens a virtual terminal for update, this is a request to set the terminal into raw mode. Once one process has set the terminal into raw mode, no other process may do so. Furthermore, the process that sets the terminal raw will now have exclusive access to the terminal until it closes it. This feature allows support of screen editors and other terminal dependent programs.

When a process calls the VT read routine, the device index is used to determine the physical port number. If this virtual open file now has the "end-of-file" bit set, that condition is returned. Otherwise a check is made to see if the file was opened for reading or for update. In the former case, a P is now done on the line semaphore for the correct device. This has the effect of suspending the process until the VTF process does the corresponding V,

indicating that an "end-of-line" character has been found. Now the characters are read out of the proper circular buffer into the EIO buffer, until the carriage return or ^D is found. If the character is a ^D, the "end-of-file" bit is also set. EIO is then told the length of the actual "block" found. This scheme of using blocked reads allows the user to employ the character and line delete features without the worry that the program will read the characters before the carriage return is struck.

When EIO calls the VT write routine, it will pass a single character since all write or update opens are character-oriented. If this character is a printing character, it will be forwarded to the low-level serial output routine directly. If it is non-printing, it will be forwarded if the terminal is in raw mode, otherwise it will be translated to a printing equivalent with a hat prepended. If the character is a carriage return and the terminal is not in raw mode, a P will be done on the hold semaphore, followed immediately by a V. Since the hold semaphore starts at Ø, this sequence produces no effect unless the ^S key has made the VTF also do a P on the hold semaphore. In that case, the outputting process will be suspended until the matching ^Q forces a V on the hold semaphore. After the P and V the carriage return is outputted along with a line feed.

The VT close and status routines are straightforward. The former just recycles the virtual open table entry while the latter returns the number of characters backed up in the circular buffer (if the virtual file was opened for reading). Terminal seeks or renames are not allowed.

The Main Body of LIZ

The re-entrant portion of LIZ expects arguments much the same as those programs designed to run under LIZ (refer to Chapter 3). The LIZ startup code will provide the top-level LIZ with standard input, output, and error output assigned to the VTF.

LIZ Operation

LIZ emits a prompt (a dash) to the standard output. LIZ then reads a line from its standard input, breaking up the line into "tokens" (lexically atomic quantities) by discarding blanks and splitting off special characters.

When LIZ encounters the in-line file character, NEWPIPE (described later) will be called to create a pipe and the pipe name will replace the in-line file character. LIZ will then take all input up the matching closing bracket (they may be nested) and write it to the pipe. A maximum in-line file size is imposed so that LIZ will not be blocked by overfilling the pipe. While LIZ is writing characters to the pipe, carriage returns cause LIZ to prompt again with a special prompt, to indicate to the user that he or she is

typing into an in-line file. When the trailing bracket is seen, the pipe is closed from the write end and LIZ continues to parse the command line. Parsing stops when the carriage return is encountered while not in an in-line file.

The first character of the input line is now checked. If this first character is a star (*), this indicates that all commands on this line are to be executed in the background. If the first character is a question mark, this indicates that all commands on the line are to be executed in trace mode. The first command on the line is then examined. If this is a built-in command, the background or trace options are ignored and the command is executed immediately. The "liz" built-in command is the only exception to this rule.

For normal commands, LIZ now scans the command line, spawning a child process for each command seen. A token is considered a command if it is either the first token on the line or if it immediately follows the ampersand (&) or pipe (|) characters. Whenever LIZ encounters the pipe character, it calls the NEWPIPE routine to build a new pipe and appends the pipe name to the pipe character in the saved command line. Each child process is spawned with a pointer pointing to the command string. If this process is to get input from a pipe, the pointer will point at the pipe sign preceding the command name, otherwise it will point at the command name.

After having spawned a child process for each command, LIZ will insert the process IDs of the children into one of two tables, the "foreground" table or the "background" table. In the former case, the table is used as input to the WAIT system call so that LIZ may suspend until one of the children finishes. When a child finishes, LIZ removes that entry from the table and calls WAIT again, until all of the processes have terminated.

When LIZ builds the foreground table, the process ID of the VTF is also inserted. If the user then employs the keyboard interrupt, LIZ will be awakened due to the VTF becoming SUSPENDED. LIZ will use KILL on each living child and then RESUME the VTF. Certain exceptions apply: LIZ will not kill off children that are copies of LIZ itself, so that they may also be awakened by the ^C. The "sub-LIZes" (invocations of LIZ by itself) will kill off their children but will not bother with RESUMEing the VTF since the top-level LIZ does that.

If LIZ has spawned one or more child processes for the background level, their process IDs are saved in the background table so that the "lj" command may reference them. LIZ does not wait on those processes, it prompts for another command immediately. The "lj" built-in command will simply read the background table, and then read the process table, indicating which background processes are still running.

LIZ's Children

The child processes that LIZ spins off are passed a pointer to the command string, the name of this LIZ's virtual terminal, and an indication of whether or not the command is to execute in trace mode. The child must first allocate a new data area, laid out in the same fashion as LIZ's data area, and copy its command string into this area. This is required because LIZ may have spawned this process for background execution and will reuse its own data area. In order to prevent a race here, the children initially run one priority level higher than the parent, but adjust their priority downward after copying their command string.

Each child processes its command string. The virtual terminal is opened as the initial standard error output, in case of syntax error messages, etc. The "<", ">" and "^" operators are interpreted, with the identifier following them used as the file name to open for standard input, output, or error output, respectively. If the pipe sign is the first character of the command string, the pipe is opened for reading (the parent will have appended a character to the pipe sign in order to name the pipe) to be the standard input. If the pipe sign appears after the command identifier, the pipe is opened for writing to be the standard output. The trailing pipe sign, or an ampersand, or a carriage return, serve to delineate the extent of the command string.

Any standard I/O not explcitly redirected is assigned to the virtual terminal. Once the standard I/O channels have been set up, the child computes the argument count. The argument count is the number of identifers in the command string including the command itself, but not including I/O redirection operators or their arguments. The BECOME system call is then employed (tracing may be turned on at this time) to actually begin execution of the named command. If BECOME returns, it is because of an error. An appropriate diagnostic is issued to the standard error output and the child EXITs.

The one time that the child process does not use BECOME is when the "liz" built-in command is the command. In that case, the child simply sets a flag in its data area indicating that this is a "sub-LIZ" and begins executing the main body of LIZ.

Pipes

Pipes are virtual devices that act as FIFO buffers between two processes. Their distinguishing characteristic is that each pipe will have two "ends" open - in other words two different file descriptors from EIO (mapping onto two different device indices), one for reading and one for writing. Pipes allow directing the output of one process to the input of another, and are especially valuable in the GOLEM environment due to the relative slowness of the secondary memory (floppy disk). Without pipes, it would be necessary

to write the output of one process to the disk and then later read it as the input of a second process.

Pipes are implemented independently of LIZ and could certainly have been implemented for a system where a command interpreter was not necessary. Nevertheless, pipes are mainly used to connect two programs running under LIZ, therefore it seemed appropriate to include them as part of the command interpreter portion of the development.

The pipe code is just a set of logical device handlers. A small initialization routine is called as part of system startup, which initializes the pipe table and performs the CONNECT. The pipe names are "|?" (recall that '?' matches any one character). In addition to the device handlers, there is one other routine called NEWPIPE, that actually creates a pipe.

As mentioned, a pipe is typically created by LIZ for two processes to use. LIZ calls the NEWPIPE routine, which finds an empty entry in the pipe table. The entry is initialized, and the index of the entry is added to the character "A", to form the second character in the name of the pipe, which is returned to LIZ. The first pipe has index zero, and is named "|A", the second has index one and is named "|B", and so forth. The child processes then open the pipe, one child opens it for reading, the other for writing.

Each pipe is allocated a page of memory for the FIFO buffer. Data flow in the pipe is controlled by two sema-

phores, and head and tail pointers. Pipes are declared as character-oriented devices to EIO.

When the device write routine is called with a character, a check is first made to see if the read end of the pipe has been closed. If so, an EIO error is returned. Otherwise the character is inserted at the write pointer and the pointer is advanced. If the pointer hits the halfway mark or the end of the buffer (in the latter case, it is then reset to the start of the buffer) a V is done on the read semaphore followed by a P on the write semaphore.

When the device read routine is called to return a character, a check is made to see if the write end has been closed. If this is the case, and the read and write pointers are equal, no more data is left and the end-of-file condition is returned. Otherwise, a check is made to see if the read pointer is at the beginning or the halfway point of the buffer. If so, a V is done on the write semaphore and then a P on the read semaphore. After this, the character is retrieved from the read pointer and the read pointer is advanced. If it hits the end of the buffer, it is reset to the beginning.

Both the read and write pointers start at the beginning of the buffer, and the two semaphores start as 1. To see how this mechanism works, let us begin with no previous reads or writes. The process that writes characters will be able to write up to halfway into the buffer before it will

become blocked by the P on the write semaphore. If the read process tries for a character, it will block on the P of the read semaphore but not before doing a V on the write semaphore. Once the write process reaches the halfway point, it will do the V on the read semaphore allowing the read process to start to read characters. This sequence continues, round and round the buffer, with each side releasing the other side with a V whenever it hits one of the halfway points, before doing a P.

When the first of the two sides does a close (this will normally be the write side, unless the reading process aborts), the pipe is marked "half-closed". At this time, both semaphores have a V operation done. This ensures that the read pointer can now catch the write pointer, or if the read side has closed, it ensures that the write side will unblock and detect the error condition. When the matching close occurs, the pipe is torn down. The pipe table entry is released, and the FIFO buffer is returned to free memory.

Seeks and renames are illegal for pipes, but the status call is honored, and the number of characters currently in the pipe is returned.

CHAPTER 9

DEBUGGING DURING SYSTEM DEVELOPMENT

The debugging of a multi-tasking operating system is difficult in the very early stages of the development. No in-circuit emulation tools were available, and the facilities of the General did not include breakpoints or mechanisms to regain control from a runaway process - other than the RESET button. The development plan called for an integrated debugger that would rely on as little of the system working as possible.

The Minimal System

The GOLEM hardware has two serial I/O ports. It was an easy decision to reserve one of the ports to be dedicated to a terminal for the debugger during the development process. The other port was used to try to run LIZ when that stage of the development occurred. The debugger was designed so that when it was invoked, it was passed the physical port number where interaction would take place with the user; the initial system start-up code told the debugger to use port 1. The only parts of the system that had to function before the debugger could be run were the kernel, the Sexton, and the low-level serial I/O drivers.

Developing the Kernel

The kernel was, of necessity, the first piece of code that had to be debugged. Due to its fundamental nature, it was not possible to use many tools to aid in debugging the kernel. This was the point where the idea of the minimal kernel paid off. The kernel was debugged by the author's repeated scrutiny of the assembly listing. Since the complete listing is less than 11 pages of assembly code, this approach was satisfactory. Initial tests of the kernel also made use of the General's ability to display memory. A test would be run, the RESET button hit, and then the process tables, etc. would be examined to see if things appeared to be working. In this fashion, it was demonstrated that processes could be created, that they would be given equal processor time (if they were the same priority), that the P and V synchronization mechanisms worked, and that trace interrupts could be generated and handled.

Developing the Sexton

Once the kernel was working, the Sexton was required in order to implement the WAIT system call. WAIT is a fundamental system call for the debugger because it must be used to notify the debugger when a traced process has hit a breakpoint. The Sexton depended on the memory allocation routines. These routines were actually debugged in the QUAD2 environment, since they did not really depend on the

QUAD3 kernel except for avoiding races. That part of the code was carefully hand-checked.

Once the memory allocation routines were done, the Sexton itself was debugged in much the same fashion as the kernel - careful desk checking and examination of memory after test runs.

Debugging the Serial I/O Drivers

The serial I/O drivers proved to be the most difficult part of the entire system to debug. The GOLEM hardware uses the 6551 serial I/O chip. Unfortunately, there are a number of "undocumented features" in the chip. After checking and re-checking the code, a logic-state analyzer and an oscilliscope were brought to bear upon the problem. Once the unusual behavior of the 6551 was understood, the code was appropriately modified and there were no more problems.

When the serial I/O drivers were working, small test programs were written to input and output large numbers of characters. These indicated very poor performance from QUAD3. Up to this point, the kernel had called the scheduler at the conclusion of every interrupt, and since many or most of the interrupts in a loaded system did not change the status of any process, the scheduler was not scheduling a new process. This constant "non-scheduling" wasted an enormous amount of the processor's cycles.

Once the problem was identified, it was a simple change to only schedule when a V operation in an interrupt handler actually unblocked a process. Of course scheduling was still done on the interval timer interrupts, to arbitrate time between CPU-bound processes of equal priority.

The Mini-Shell

With the kernel, Sexton, and serial I/O drivers in place, it was possible to have the debugger on one of the serial I/O ports. The debugger was called the "mini-shell", the term "shell" being borrowed from UNIX. The QUAD3 initialization portion, which is the code that defines a particular QUAD3 configuration, starts the mini-shell for the second port. The mini-shell itself had to be debugged in the same fashion as the kernel and Sexton.

The mini-shell offers a basic set of commands to: print and change memory, "acquire" a stopped process, single-step a process, and display the registers of a stopped process. Three more commands were added to test the floppy disk handler. They were: display the sector buffer, read a sector into that buffer, and write a sector from that buffer.

Since the mini-shell did not depend on the EIO file system, or the command interpreter, all of the rest of QUAD3 could be developed with the aid of it. This included LIZ, the VTF, the EIO system, the QUAD2 compatible file system driver, the line printer driver, portions of the memory

manager dealing with module and soul management, the pipe driver, and several small user programs to use in testing LIZ.

The mini-shell was written in re-entrant code. This was initially used to demonstrate the true multi-tasking nature of QUAD3, by running the mini-shell simultaneously on both I/O ports. Once QUAD3 and LIZ were debugged, it was no longer necessary to initialize the mini-shell on one I/O port. Since it was re-entrant code, the mini-shell was retained (with minimal changes) as a subroutine of LIZ, accessible via the "debug" built-in command.

Debugging User Programs Under LIZ

When a program is being developed to run under QUAD3, the user may take advantage of all of the debugging services of the mini-shell at any time. He may insert fixed breakpoints in his code at assembly time with the software interrupt instruction if so desired. In that case, when he invoked the program from LIZ and it hit a breakpoint, LIZ would print a message indicating the process number and the fact that it stopped at a breakpoint. The user can then enter the mini-shell with "debug".

Another way to debug a process is to start the LIZ command line with '?'. The invoked program will then start in trace mode, thus it will stop (and LIZ will print the same message) right after the first instruction of the test program.

Once the mini-shell is entered, the user then acquires the process to be traced. Note that more than one process may be stopped in trace mode. For example, the user may debug two commands joined with a pipe with '?'. The user may trace only one process at a time, but may switch processes at any point. When a process is acquired, the registers are displayed. The registers are redisplayed each time the process is stepped or hits a breakpoint. The user may also release a process at any time. If a released process hits a breakpoint, it will be suspended until reacquired by the mini-shell.

The mini-shell competes for terminal I/O with processes being debugged. In order to resolve this problem, LIZ advances the mini-shell's priority higher than that of the VTF. Thus programs that do console I/O can only do it while the mini-shell is waiting for a process to hit a breakpoint. In practice, this works well and avoids interlaced terminal input and output.

CHAPTER 10

CONCLUSIONS

The support software described in Chapter 2 was finished by the end of 1980. Most of the design of QUAD3 had been decided on, but the actual coding was done on weekends during 1981 and 1982. The coding and debugging went rapidly, especially after the mini-shell was working. By October 1982, QUAD3 was fully functional. This included both the operating system proper, the QUAD2 compatible file system driver, LIZ, and miscellaneous drivers for things like the line printer. The reason for this rapid progress was the large amount of groundwork. The macroprocessor, relocating assembler, linker, and QUAD2 system for development, proved to be an excellent software development setup.

QUAD3 already has a programmer's manual that outlines the system calls and conventions. It is presented in Appendix A of this document. A QUAD3 user manual will appear when more QUAD3 commands (programs that run under LIZ) are available.

QUAD3-Compatible Programs

As of this writing, QUAD3 has few programs written for it because it is new. A few programs were written for test purposes during the system development. These include the "wc" and "tr" programs mentioned in Chapter 3. There is

a program to display files on the CRT. There is also a program to list a QUAD2-compatible disk's directory (written by T.B. Williams) and a program to convert tabs in standard input to the appropriate number of spaces in standard output for devices that do not support tabs (written by R. Milakovich). The author also wrote a program to compute the first 1024 prime numbers, to be used for benchmarks.

The most significant program now available under QUAD3 is the macroprocessor. MACP was originally written with QUAD3 in mind, and we decided to write it using QUAD3 system calls. It was brought up and used under QUAD2 with the addition of a software package that translated the formats of I/O system calls, and simulated QUAD3-type memory allocation. When QUAD3 and LIZ were stable, the macroprocessor was then ported. This took about 30 minutes from starting until the job was complete - a resounding success in our estimation.

Future plans call for the relocating assembler and linker to run under QUAD3. The primary problem in converting is to change those areas of the programs that manage a symbol table in a fixed address, so that they can use dynamically allocated memory instead. The QUAD2 screen editor may also be ported over. In this case, the edit buffer would be a static data area at the end of the program. This approach would simplify the port, at the expense of having the editor hog memory, even for very small files.

QUAD3 Performance Characteristics

A small demonstration program was written to test various aspects of QUAD3 performance. It is a program to print the first 1Ø24 primes. The algorithm was Knuth's (1973) algorithm P. The program was designed to work under LIZ as the "primes" command. Primes prints out each prime on a separate line (to standard output) as the prime is discovered.

The program was run and it printed out the first 1Ø24 primes on the CRT in 63 seconds. An option was then added to delay printing until all of the primes were computed. This also ran in 63 seconds. This was somewhat surprising since QUAD3 has a 64 byte output buffer for serial output. This output buffer would hold about 13 four-digit numbers (each also had a carriage return). The conclusion to be drawn was that the program could generate primes much faster than the 96ØØ baud CRT could display them, and that the savings in time from having the first 13 or so lines buffered up were negligible.

To verify that this program was indeed I/O bound, an option was added where standard output was not written. This version ran in 49 seconds. This showed that combined overhead of the EIO system, and the slowness of the CRT was 14 seconds. Next, the program was run with standard output directed to the null device. The time was 54 seconds, which indicated that the EIO system overhead was 5 seconds. Since

the total number of characters output was roughly 5000, this indicates an EIO overhead of about 1 millisecond per character, which is very good (about 300 instructions).

The program was then run with the standard output directed to a disk file. This time it took 67 seconds, excellent for a floppy disk with a linked-list type of file system.

The multi-tasking features of the system were tested by running two copies of "primes" simultaneously, both outputting to the CRT. This took only 124 seconds, less than twice the 63 seconds when only one copy would run. The savings was due to the fact that while one process may be waiting on the CRT, the other might be running. What was not known was whether more savings were offset by the additional overhead of switching between the processes. In order to estimate this overhead, two copies of the program were run simultaneously with the "no output" option. This took 98 seconds. Since the single case had taken 49 seconds, the context-switching overhead from interval timer interrupts was clearly negligible. This was seen as an endorsement for the minimal kernel approach.

Comparing QUAD3 With QUAD2

There is a natural desire to see how QUAD3 compares with the system it was designed to upgrade. The macroprocessor was already available as a program to do comparisions. MACP was run on a 300 line file that had an

"include" of maclib (Appendix B). When MACP was already loaded in memory, QUAD2 ran the program in 35 seconds while QUAD3 ran it in 38 seconds. Since this was a single task, it was very gratifying to have QUAD3 less than 10 percent additional overhead with all of the multi-tasking logic.

QUAD3 actually shows an improvement when loading the relocatable object file with MACP. It can do the load in 29 seconds, while QUAD2 takes 34 seconds. Either one is rather long for the user to wait to begin his command. Of course, QUAD3 remembers if MACP has been loaded and will reuse it if possible. QUAD2, on the other hand, does not have to worry about relocation, thus the normal method of loading MACP is to load an absolutely originned version using the "load tracks" mentioned in Chapter 1. This takes only 5 seconds.

QUAD3 cannot compete with QUAD2's load tracks. Only if the 6502 supported position independent code, and if QUAD3 were willing to give up the complete device independence it now has, could a similar facility could be provided.

The "tr" and "wc" programs were converted to run under QUAD2 for additional benchmarks. These were chosen because they are I/O-bound and make use of line and character I/O respectively. After running both programs under each system, it was found that the QUAD3 versions ran about 20 percent slower.

Overall, QUAD3's performance was considered excellent, since it was doing a single task and being matched against a single-tasking system. Unfortunately, there is no real way to directly benchmark the two systems where QUAD3's true advantage, multi-tasking, comes into play - because QUAD2 can't do it !

QUAD3 Concepts and UNIX Concepts

Many of the user interface ideas in LIZ came from the UNIX shell (command interpreter). No attempt will be made here to compare the two systems, since UNIX does not run on a very small computer like QUAD3, but certain differences should be discussed insofar as they affect the user interface.

Both systems allow the user to run commands in the "background" and to report on the status of those commands. Both systems can spin off another copy of the command interpreter to execute files of commands. The difference is that when LIZ does the "liz" command with input redirected from a command file, each command in the file must specify input redirection as well, unless it is to get input from the terminal. The same goes for redirection of output. This is because UNIX allows more than one process to read or write from a file, thus the standard input and output of the spun off command interpreter can also be the default standard I/O of commands it does. This is clearly preferable.

QUAD3 offers some solace with the built-in file feature. Consider the following UNIX command file:

```
ed test
1,$s/foo/bar/g
w
wc < test > test.size
```

This file calls the UNIX editor, and substitutes "bar" for every occurrence of "foo". The changed file (test), is written out, and then the words are counted and put in the file "test.size". If the file name were "cmd", it would be run in the background with "sh cmd &".

Assuming that QUAD3 had the same editor, a similar function could be done with "*liz < cmd" if the file "cmd" had:

```
ed test < {
1,$/foo/bar/g
w
}
wc < test > test.size
```

Here the in-line file has made up for the inability of the child process to read its parent's standard input file.

UNIX is unabashedly directed towards support of terminals, unlike pure QUAD3's pretensions of generality. Because of this, each process in UNIX can be associated with a I/O port. This makes the handling of keyboard interrupts much easier, each process associated with the terminal is given a "signal" when the keyboard interrupt is struck.

QUAD3 does not assume any terminal, thus all terminal dependent functions were pushed into LIZ and the VTF. This created all sorts of "warts" in order to have the

keyboard interrupt only affect processes that "belonged" to the person who struck it.

UNIX enforces a standard format on all file systems associated with it, whereas QUAD3 does not. In QUAD3, any sort of file system may be supported as long as the appropriate driver is connected. This does complicate issues such as listing directories, since there must be a separate command for each file system. The usual solution is to have the directory lister reside on the file system that it is familiar with.

In retrospect, QUAD3 benefitted most from the UNIX concepts of standard input, output, and error output and from the idea of "pipes". The whole idea of "filters" was carried over directly, and this proved to be a QUAD3 strong point.

Summing Up

The ideas for QUAD3 were first formulated back in early 1979. At that time QUAD2 was barely functional, processors like the 68000 did not exist, and there were no multi-tasking operating systems for microcomputers. In Fall of 1979, the author became employed with a commercial firm, thus the later development of QUAD2 and all of QUAD3 was accomplished on weekends. While QUAD3 has grown up and met almost all expectations (unusual for a software project !) the world at large has not stood still.

The 6502 processor is now obsolete. Several multi-tasking operating systems are available for the newer micro-computers. To the author's knowledge though, no one else has ever successfully implemented a general multi-tasking operating system on any primitive CPU like the 6502.

The 6502 implementation of QUAD3 then is both remarkable, and somewhat dated. The GOLEM hardware is slated for eventual replacement by systems with the newer processors. The basic concepts of QUAD3 though, are as topical as ever. In particular, the idea of the minimal kernel, the ability to dynamically install interrupt handlers, and QUAD3's best idea: dynamic connection of virtual device drivers, have been proven sound and are applicable on the newer systems as well.

APPENDIX A

This appendix is a direct copy of the QUAD3 Programmer's Manual, used by people who wish to program under QUAD3. As of this writing, the utility of the manual has been demonstrated by having two people successfully implement QUAD3 commands with no other guide than the manual.

Introduction

QUAD3 is a multi-tasking operating system designed to run on the GOLEM microprocessor system, much like its predecessor, QUAD2. Like QUAD2, QUAD3 is still oriented towards a single user per system, but the single user may have several programs running at once. For example, the user may have an assembly running output to the line printer and be editing another file while the assembler is running.

The QUAD3 command interpreter, called LIZ, is just another program running under QUAD3. Initially, QUAD3 is set up with LIZ running for port Ø - In other words, QUAD3 will have a command interpreter on the same CRT as would QUAD2. It is possible though, to run another LIZ for port 1, which would allow two people to use the system simultaneously ! All "commands" to LIZ (except for a few built into LIZ) are just the names of files. LIZ takes the name, loads the file, and executes it.

QUAD3 does not assume any particular I/O setup, devices are added dynamically to the system and so are their driver and interrupt routines. For convenience, QUAD3 is typically "genned" so that certain standard devices are present when the system is booted. In particular, the terminals, the line printer, the null device, and the QUAD2 compatible floppy disk system are declared. The user may add other devices or different handlers for declared physical devices (e.g. a disk driver for a file system other than QUAD2's).

In the following sections, the LIZ command interpreter, QUAD3 programming conventions, the QUAD3 system calls, and the procedure for writing LIZ-compatible programs will all be laid out. There will also be sections dealing with how to write device drivers, and how to write interrupt handlers.

Booting QUAD3

QUAD3 must be read into memory while in the GNL console monitor. If QUAD3 resides on a QUAD-style load track on the floppy disk, the following procedures should be followed:

1) Set the RAM-protect switch to "ON".
2) Type "L QUAD3" to the console monitor.
3) Set the RAM-protect switch to "OFF".
4) Type "G" to the console monitor.

LIZ should be initialized on port Ø. This will be indicated by a dash ("-"), LIZ's prompt. In addition, the

lowest of the 8 LEDs on the processor board will blink, indicating that the operating system is running.

The blinking light is produced by the null process. The top four LEDs indicate the four least significant bits of the process number currently running. If these bits are stuck to one value and the lowest LED does not blink, the process in question is using all of the processor cycles. This may be OK, if the process does a lot of computational work, otherwise it may indicate the dreaded illegal instruction.

LIZ - The QUAD3 Command Interpreter

"LIZ" is short for "Lizard Monitor", the full and correct name of the user interface to QUAD3. LIZ allows one to run one or more programs, possibly connecting some program's outputs to other program's inputs, in either the "foreground" (LIZ waits for completion) or the "background" (LIZ starts the program(s) and prompts for more work).

LIZ prompts with a dash. The user then types in a command string, which is terminated with a carriage return. In general, no action is actually taken until the carriage return is hit, but certain control characters are exceptions: The ^E or RUBOUT characters will delete the last character entered. The ^U character will delete the entire line. The ^C character will kill any programs executing in the foreground.

LIZ views the command line as a series of tokens. A token is a string of characters separated by white space or special characters. In order to include blanks, tabs, special characters (defined later) in a token, it must be surrounded with double quotes. LIZ takes the first token as the command, and other tokens up to a special character as arguments. The command will be executed, passing the arguments verbatim.

There are two exceptions to the above rule and they deal with the first character of the command line. If the first character is a star (*), it will not be considered part of the first token, but rather it indicates to LIZ that all of the commands on this line should be run in the background - in other words, they will be started by LIZ but LIZ will come back for more commands while they are still running. Similarly, if the first character is a question mark, it specifies that all of the commands on this line are to be run in trace mode. For more on trace mode see the QUAD3 user's manual.

There are certain concepts associated with the programs that run under LIZ. These notions are of "standard" input, output, and error data streams. These data streams are like pre-opened files that many programs may read from or write to. Since programs that run under LIZ often use these constructs, it is possible to direct the input and output of programs when they are invoked by LIZ, and some-

times possible to hook two programs together. The special characters are used to specify I/O redirection as follows:

1) The ">" character means direct the output of a command to the file name which is the following token.

2) The "<" character means direct the input of a command from a file name (the next token).

3) The "^" character means direct the error output of a command to the file specified by the next token.

4) The "|" character means take the output of a command and make it the input to another command. The second command string starts after the "|".

5) The "&" character ends one command string and specifies another command to be run at the same time.

Where we have mentioned tokens that represent filenames, a device name may serve as well. There is no distinction made between files and devices in QUAD3. In fact, every file name contains an implicit device name. Simple names with only alphabetic characters have the null string for a device name (do not confuse this with the null device, whose name is ":"). The null string is the name of the QUAD2 compatible file system on the floppy disk. Thus most commands will come from that file system. It is perfectly legitimate to load a command from another device though.

One more special construct is possible. Wherever LIZ expects a file name (for reading only), the user may use the "{" special character. This indicates the beginning of an "in-line" file. All text following the opening brace is

read without parsing and becomes available as the input just as if it were in a named file. The end of the in-line file is marked with a "}". The closing brace may not be hidden with double quotes since the text is not parsed, but braces may nest, with only the outermost pair considered special. The in-line file may contain carriage returns and LIZ will prompt with "->" after each return, as a reminder that an in-line file is being input.

a) -cmnd1 > file1

b) -cmnd1 | cmnd2 | cmnd3 > file2

c) -cmnd1 < {text....text} &

The examples above show some typical uses of LIZ. In example a, a command is run with the output sent to a file called "file1". In example b, the output of the first command is the input to the second command, which in turn has its output sent as the input to a third command. The third command has its output sent to a file. In example c, a command is run in the background with an in-line file as input. In all cases where the input, output, or error output is not explicitly directed, the terminal will be the default. Note then, that example c might produce terminal output if cmnd1 produces output. Since this is running in the background, the terminal output may appear while the user is doing something else.

LIZ has a few built-in commands. The "liz" command causes LIZ to execute a copy of itself as a command.

Typically this is done in conjunction with input redirection:

```
-*liz < {cmnd1
->cmnd2
->}
```

In the example, note LIZ's special prompt while the user types in an in-line file. This example shows how the user can have LIZ run two commands in sequence in the background. Notice how this differs from:

```
-*cmnd1 & cmnd2
```

In the second example, both commands run in the background too, but simultaneously.

Another command is the "kill" command. This command takes a process ID and terminates the process. LIZ will display all background processes (run by this LIZ), their IDs, and their status with the "lj" command.

The "lm" command displays the names and addresses of all of the programs currently loaded into memory along with the number of processes running for each. The "debug" command is used in conjunction with trace mode. It invokes a resident debugger capable of tracing programs under test. For more on this, see the QUAD3 user's manual. For other commands (not built into LIZ) typically available on QUAD3 systems, see the QUAD3 user's manual also.

Writing QUAD3 Programs

QUAD3 programs must be relocatable. This means that they must have been assembled with the relocating assembler

and if linked, must not have had the -a, or -d options in the linker.

The meager 6502 instruction set is available, with the exception of the SEI instruction. Accessing absolute locations (except for memory-mapped peripherals) is forbidden, since the program is given memory at run time. The only locations in the zero page that may be used are the "registers", locations ØØ - ØF. The last two will be initialized to point to the end of a "user stack" which every process must have if it wishes to use system calls. The user stack is a convenient place to save and restore the zero page registers via system calls such as PUSH, PUSHP, PUSHALL etc.

Data areas may be statically allocated as assembler DSECTs or may be hidden in PSECTs, but they must be relocatable. There is a system call (ALLOC) to get additional memory at run time. If the program is to be re-entrant, only dynamically assigned memory or the registers can be used for data storage.

The only other requirement for programs that run under QUAD3 is that they have a "load module header" at the start of the program of 18 hex (24 decimal) bytes. This header should be zeroed except for the last byte which may have bit 6 set if the program is reusable. A reusable program is one that may be re-executed without a reload. In general, programs are reusable if they do not use the load

to initialize data variables. If bit 7 is set, this indicates that the program is re-entrant - the one code body may support multiple processes. All re-entrant programs would have to be reusable.

When the program is loaded, the rest of the load module header area is initialized for system use. Control is then passed to the code immediately following the header. The program may now go its merry way, bearing in mind the restrictions stated above plus the additional caveat that the program may not have its hardware stack pointer go below D6 hex, due to context-switching space limitations.

System Calls

The QUAD3 multi-tasking operating system offers a variety of system calls to allow the user to perform I/O, allocate memory, manage his user stack, and control processes. The system calls are all accessed via a "JSR" instruction to the external name of the system call. When the program is linked, the "-p" option of the linker will resolve the addresses. The file "predef.o" is required.

The following conventions apply to all system calls except where explicitly denied:

1) The program must have at least 128 bytes available on the user stack and at least 2Ø bytes available on the hardware stack (SP no less than EA hex).
2) All registers (RØ-R15) are preserved, as well as X and Y.
3) A contains a call-dependent return status (often junk unless V set)
4) The V bit is used to indicate errors

5) The C bit is used to indicate end of media type status (only if no bytes fetched)
6) The I bit will stay clear
7) Other bits in the status register are random, except the D bit will be guaranteed clear if it was clear when the system call was made.

Extended I/O System Calls

These system calls often deal with an entity called a file descriptor. The file descriptor is an eight-bit quantity returned from the OPEN system call which is used for all further communication with the system about that file.

The following standard return codes may be returned in A if the V bit is set:

-1	Unspecified I/O system failure
-2	Device hardware failure
-3	Bad file descriptor
-4	Device limitation (e.g read from line printer)
-5	Device not ready (e.g disk door open)
-6	File not found
-7	Overwrite protection
-8	Invalid file name
-9	Invalid device name
-10	Out of buffers or file space
-11	Invalid access (e.g. write on file opened for read)
-12	Device busy

The IOERR system call may be used in conjunction with these error codes to reproduce the above English messages.

NAME: OPEN - Open a file

INPUT:

```
P0 -> filename
A = OP_RD  (0) for reading
  = OP_WRT (1) for writing
  = OP_UPD (2) for update
```

OUTPUT:

```
Y = file descriptor
```

DESCRIPTION:

The OPEN system call makes a file (or device) available for reading, writing or updating (both reads and writes). The returned file descriptor is used for further system calls having to do with the file. It is valid until the file is closed. If an existing file is opened for writing, it will be truncated to zero length initially.

NAME: CLOSE - Close a file

INPUT:

```
Y = file descriptor
```

DESCRIPTION:

The CLOSE system call removes the binding between a file descriptor and a file. It is possible to get error messages from any final writes that CLOSE does to flush the buffers, so errors should be checked for.

NAME: CLOSALL - Close all files belonging to this process

DESCRIPTION:

A CLOSALL occurs automatically when a process dies, thus a process need not explicitly close files. Nevertheless, open files do consume resources, so they should be closed explicitly when the process is through with them.

NAME: READ - Read data from a file

INPUT:
        Y = file descriptor
        PØ -> buffer area
        P2 = requested count
OUTPUT:
        P2 = actual count

DESCRIPTION:
     The READ system call requests that a number of bytes of data be read from the file. The amount read will be the amount requested unless "end-of-file" is encountered first. The read data will be stored in the buffer area pointed to by PØ.

NAME: READL - Read line from file

INPUT:
        Y = file descriptor
        PØ -> buffer area
        P2 = maximum count
OUTPUT:
        P2 = actual count

DESCRIPTION:
     READL is identical to read except that data is not read past a carriage return (ØD hex). If a carriage return is not encountered before the maximum count is reached, then the read stops at the maximum.

NAME: READC - Read character from a file

INPUT:
        Y = file descriptor
OUTPUT:
        A = character

DESCRIPTION:
     The READC call is a quick way to get a character from a file without setting up a buffer.

NAME: WRITE - Write data to a file

INPUT:
        Y = file descriptor
        PØ -> data
        P2 = amount to write
OUTPUT:
        P2 = amount written

DESCRIPTION:
     The WRITE system call allows writing a block of data to a file. P2 will be the same on output unless an error has allowed only a partial write.

NAME: WRITEL - Write data (Stop on null)

INPUT:
        Y = file descriptor
        PØ -> data
        P2 = amount to write
OUTPUT:
        P2 = amount written

DESCRIPTION:
     The WRITEL system call is identical to write except that it will stop if a null is encountered in the data. This is useful for outputting canned strings, etc. without counting their length.

NAME: WRITEC - Write a character to a file

INPUT:
        Y = file descriptor
        A = character

DESCRIPTION:
     This call is a quick way to write a single character to a file.

NAME: SEEK - Move file pointer

INPUT:
        Y = file descriptor
        P2 = offset
        A = SK_ABS (Ø) for absolute offset
           = SK_REL (1) for relative offset

DESCRIPTION:
     The SEEK system call allows changing the "file pointer", the point where the next byte would be read or written, of a file. Relative seeks are from the current position, P2 is taken as a 16 bit signed number. Seeks off either end will stop at the end or beginning of the file.

NAME: STATUS - Get status of a file

INPUT:
        Y = file descriptor
OUTPUT:
        P2 = current file pointer
        P4 = size of file
        A = ST_EOF (8Ø hex) and C set
           * OR *
        A = ST_NEOF (Ø) and C clear

DESCRIPTION:
     The STATUS system call gives information about the file. It is mostly used for finding where the file pointer is (this will be Ø for terminals) or for testing before reading to prevent a block.

NAME: RENAME - Rename or delete a file

INPUT:
        PØ -> Old filename
        P2 -> New filename (if null, delete old file)

DESCRIPTION:
     The RENAME system call is the only system call besides OPEN that does not use file descriptors. A file does not need to be opened to be renamed or deleted.

NAME: PUTHEX - Output two hex characters

INPUT:
A = Hex number to output
Y = File descriptor
OUTPUT:
A = Standard I/O error codes

DESCRIPTION:
This system call is a shorthand for a frequent coding sequence where a number is to be converted to hex and then written to some file.

NAME: IOERR - Get I/O error string

INPUT:
A = Error code from extended I/O
OUTPUT:
PØ -> null terminated error message

DESCRIPTION:
This call may be used to get canned explanations for the various I/O errors. If A is not a valid code, the string will be "Invalid error code".

System Calls to use the User Stack

These system calls are provided to facilitate saving data on the user stack. The QUAD3 hardware stack is small, because it must be copied in and out every time a process runs. As a result, the user stack is expected to bear the brunt of saving registers etc. for subroutine calls. Every process is given a 1 page user stack when it is initialized. A larger user stack may be had by calling ALLOC with the desired number of pages, and then switching P14 (the usual name for location ØE, also called sp) to point to the <u>end</u> of the acquired memory. The old page should be freed with FREE.

NAME: PUSH

DESCRIPTION:
Accumulator A is saved. (sp) will now point to the saved A.

NAME: PULL

DESCRIPTION:
Accumulator A is restored.

NAME: PUSHXY

DESCRIPTION:
The X and Y registers are pushed. Y is pushed first so (sp) will point to the saved X.

NAME: PULLXY

DESCRIPTION:
The X and Y registers are pulled.

NAME: PUSHPØ

DESCRIPTION:
Register pair PØ (locations Ø and 1) is pushed. The MSB (R1) is pushed first.

NAME: PULLPØ

DESCRIPTION:
Register pair PØ is pulled.

NAME: PUSHP - Push pair

INPUT:
X -> register pair to push

DESCRIPTION:
The specified pair is pushed, MSB first.

NAME: PULLP - Pull pair

INPUT:
X -> register pair to pull to

DESCRIPTION:
The specified pair is pulled.

NAME: PUSHALL

DESCRIPTION:
X,Y and all of the registers (except P14) are pushed. First Y is pushed, then X, then R13 through RØ. (sp) will thus point to the saved RØ.

NAME: PULLALL

DESCRIPTION:
X,Y and all of the registers (except P14) are pulled.

NAME: PULLSOME

INPUT:
X -> first register to be restored

DESCRIPTION:
This call pops the stack like PULLALL, but all registers less than (X) are not restored, the pulled value is discarded. X and Y are always pulled.

Process Control System Calls

These system calls relate to starting, stopping, and delaying processes. The term "PID" means process id - a 16 bit quantity used to identify a process to the system.

NAME: P - Semaphore primitive

INPUT:
X -> Register pair -> Semaphore

DESCRIPTION:
The "P" operation on a semaphore does the following: If the semaphore is less than 1, it is incremented. If the semaphore is 1, the process is blocked until the semaphore becomes less than 1, whereupon the semaphore is incremented.

NAME: V - Semaphore primitive

INPUT:
X -> Register pair -> Semaphore

DESCRIPTION:
The semaphore is decremented - If it was 1, the highest priority process waiting to do a P is now awakened. A semaphore may be used to control a resource so that only one process may seize it at a time. The semaphore should be initialized to Ø. Every process that tries to access the resource should do a P first - this will block the process until it gets the resource. When the process is done with the resource, it should do a V.

NAME: SPAWN - Start a child process

INPUT:
P2 -> Entry point for child process
OUTPUT:
A = Ø for success, -1 for failure
PØ = Childs PID

DESCRIPTION:
The SPAWN system call is the standard way to bring a new process into being. It will succeed unless there are too many processes in the system. The child process will begin executing at the address in P2. All of its registers will be the same as the parent's registers, except PØ will be the parent's PID. The child will have a user stack all set up.

NAME: SUSPEND - Stop a process

DESCRIPTION:
A process may SUSPEND itself. It will remain blocked until another process uses the RESUME system call to unblock it.

NAME: RESUME - Resume a stopped process

INPUT:
P0 = PID of process to resume
OUTPUT:
V set if process not found or not suspended

DESCRIPTION:
This call will restart a suspended process.

NAME: EXIT - Terminate this process

INPUT:
P0 = Process termination status

DESCRIPTION:
This system call is the way for a process to terminate. The termination status is provided so that processes waiting on this process may be informed of it. It may be any 16 bit number except the MSB may not be FE or FF. All files are automatically closed, and all process memory is returned.

NAME: KILL - Kill off a process

INPUT:
P0 = PID of process to murder
OUTPUT:
V set if no such process

DESCRIPTION:
This call will result in the immediate and irrevocable death of the victim. The exit code for the killed process will be FF00.

NAME: WAIT - Wait for processes to suspend or die

INPUT:
P0 -> Table of 4 byte entries: each entry has two bytes of PID and a two byte result area
A = number of entries

OUTPUT:
P0 -> Entry that terminated the wait. The result area will have a 16 bit status code.

DESCRIPTION:
This system call allows a process to suspend until one of a set of specified processes changes state. Typically, these will be children of the waiting process. The result fields provided for in the table will be random except for the entry describing the process that has changed state. If the process in question has suspended, the result entry will be FE05. If it has hit a breakpoint, the result will be FE03. If it has died, the result will be the 16 bit code the process specified when it did an EXIT. If WAIT is called for a non-existent process, it will return immediately with the entry pointed out and with the result field being FE80. Other possible exit codes may come from kernel-initiated deaths (the process did a bad thing). These are documented in "kerequ.h".

Miscellaneous System Calls

NAME: BECOME - Change program

INPUT:
P0 -> Filename
A = 0 for normal case
A > 0 to start new program in trace mode

OUTPUT:
(Normally doesn't return)
A = -1 for bad data in load file
= -2 for not enough memory space
= -4 for unable to open file

DESCRIPTION:
This system call allows a process to shed its old program code and begin executing another program. The specified file is loaded (unless it is already in memory) and the process begins executing at the load module entry point. If A is non-zero, the process will immediately block on a trace interrupt. If no other processes are executing in the old code body, it will returned to free memory.

NAME: ALLOC - Allocate memory

INPUT:
A = Ø (only valid user option)
RØ = # of contiguous pages to allocate
OUTPUT:
PØ -> start of allocated memory
A = Ø for success N clear
* OR *
A = -1 for success N set

DESCRIPTION:
This system call allows the user to get chunks of free memory in page increments.

NAME: FREE - Return memory

INPUT:
PØ -> first page to return
A = # of pages to return
OUTPUT:
A = Ø for success
A = -1 if memory was not owned by caller

DESCRIPTION:
This system call allows the user to return previously allocated memory. All allocated memory is automatically collected when a process dies, so this call is unnecessary if the process is about to terminate.

NAME: GETHEX - Get hexadecimal number from a string

INPUT:
PØ-> null-terminated ASCII string
OUTPUT:
PØ = converted number
A = number of digits converted
* OR *
C set if no hex digits found
PØ preserved

DESCRIPTION:
This system call is useful for extracting hexadecimal values from input strings. The string may have non-hex characters in it and the scan will stop on them. C will be set if the first character in the string is not hex. Upper or lower case a-f are permissible.

Writing Programs to Run Under LIZ

Programs that are called from the command interpreter are passed certain arguments which they may employ. If a program's input or output is to be directed by LIZ, the following passed file descriptors should be used: Register 8 has the file descriptor for the "standard input". Register 9 has the file descriptor for the "standard output", and register 1Ø has the file descriptor for the "error output". These are pre-opened files, hence the file descriptors. They should be treated like any other open file. Standard definitions for LIZ parameters are available in the file "liz.h".

The arguments to the program (from the command line) are available. Register 11 is the number of arguments. This will always be at least one since the program name is considered the first argument. Register pair P12 points to the arguments. Each one is a null-terminated string, and they are contiguous. The GETARG subroutine is available to facilitate scanning the arguments.

NAME: GETARG - Get next argument

INPUT:
P12 -> Current argument
OUTPUT:
P12 and PØ -> Next argument

DESCRIPTION:
This system subroutine is actually part of LIZ, but is available for user programs. If called initially, it will set PØ-> the first argument to the program, since P12 points initially to the program name.

Terminals controlled by LIZ appear in the device hierarchy as "=n" where n is the port # in ASCII. They may be opened like any file. Certain programs that run under LIZ (such as a screen editor) may want to access the terminal in raw mode (no special processing for control characters). This may be accomplished by opening the terminal for update. The terminal name is passed in registers 5 and 6, with register 7 containing a null - thus it may be opened by pointing PØ at register 5 and calling OPEN.

Adding I/O Drivers

QUAD3's unique, flexible I/O structure allows adding devices, which may have file systems, at any time. A device may support files or not, as it chooses. It is made known to the system by the CONNECT subroutine where it specifies a match string to match file/device names. Thus when the extended I/O system is given a device/file name to open, it finds the connected device whose match string matches the name. Match strings may contain any printing character, with the "*" and "?" characters taking a special meaning. It is suggested that LIZ special characters not be used in device match strings as that would force the user to quote that device name.

The "?" character matches precisely one alphanumeric character, while the "*" character matches Ø or more alphanumeric characters. (An alphanumeric character is a-z, A-Z, "." or "`".) These "wild cards" allow devices to

support file systems since the match string will now match many names.

NAME: CONNECT - Declare a new device

INPUT:

```
        PØ -> Device match string
        P2 -> ptr -> Device open
              ptr -> Device close
              ptr -> Device read
              ptr -> Device write
              ptr -> Device seek
              ptr -> Device status
              ptr -> Device rename
```

OUTPUT:

```
        X = Device #
```

DESCRIPTION:

The CONNECT routine declares a new device to the extended I/O system. The match string should contain at least one non-alphanumeric character if it is not to be the default device (typically the QUAD2 compatible floppy). P2 points to a table of device routine addresses. If any of these addresses are Ø, it indicates that the operation is not supported on the device (e.g. rename is useless unless the device supports a file system). Following are seven descriptions of the device routines that must be written for the device.

NAME: Device open

INPUT:
P0 -> device/file name
Y = OP_RD (0) or OP_WRT(1) or OP_UPD (2)
OUTPUT:
V set with A = error code
* OR *
Y = file index
A = 256 - block size

DESCRIPTION:
The device open call will be called from EIO (extended I/O) with P0 pointing to the string that matched the device match string. It is expected to return V set with A = the appropriate error code if there is an error. Otherwise, it should return Y = an index to use in further communication for this file (this might always be the same if this is a device with no file system). The file index could be described as a local file descriptor. The accumulator should have 256 - the block size. The special case of 255 is referred to as "character I/O" and it changes the input/output specs of some of the other device calls.

NAME: Device close

INPUT:
Y = file index
OUTPUT:
A = error code if V set

DESCRIPTION:
This is the device level close. The driver should free all resources used by the open file. For blocked I/O (all I/O not called "character") when writing, EIO will already have called the device write with the residual data, so the device close need not worry about that.

NAME: Device read

INPUT:
        Y = file index
        PØ -> buffer area { block I/O only }
OUTPUT:
        A = character { character I/O only }
        Y = end of buffer offset { block I/O only }
         * OR *
        C set for EOF ("end-of-file")
         * OR *
        V set with A = error code

DESCRIPTION:
     EIO calls the device read routine to get the next character or block from the file. For block I/O, the result should be right adjusted in the buffer area (buffers are one page) pointed to by PØ. The returned value in Y will then be Ø unless only a partial block was read (such as at EOF).

NAME: Device write

INPUT:
        Y = file index
        X = character { character I/O only }
        PØ -> buffer { block I/O only }
        X = end of buffer offset { block I/O only }
OUTPUT:
        A = error code if V set

DESCRIPTION:
     For block I/O, the data will be right adjusted in the buffer as though a full block was being written, even though only a partial block may be written at EOF. The block buffer is always 256 bytes long and it will not change from write to write (or read to read), therefore, the device driver may keep information in the front of the buffer if the blocksize is less than 256. This is quite useful for QUAD2-type chain pointers.

NAME: Device seek

INPUT:
        Y = file index
        X = SK_ABS (Ø) or SK_REL (1)
        PØ -> block buffer { block I/O only }
        P2 = seek offset
OUTPUT:
        X = offset of next character { block I/O only }
        Y = end of block offset { block I/O only }
        C set if seek would have gone past beginning
              or end of file

DESCRIPTION:
     The device seek routine is responsible for setting the "file pointer" so that the next character read (or written) will be at the given point. If it is block I/O, the appropriate block must be inserted in the buffer (Y should be the eob offset, analogous to the device read) and X should indicate the buffer offset of the character where the file pointer is now.

NAME: Device status

INPUT:
        Y = file index
OUTPUT:
        P4 = file size
        C set if at end of file

DESCRIPTION:
     The device status routine may return a length of Ø for files such as the terminal, if no character is presently available. EIO will use the C bit to decide whether or not the logical "end-of-file" has been reached.

NAME: Device rename

INPUT:
        PØ -> old name
        P2 -> new name
OUTPUT:
        A = error code if V set

DESCRIPTION:
     Device rename should only be supported if the device has a file system. If name 2 is null, the specified file should be deleted.

NAME: DISCONNECT - Remove a declared device

INPUT:
X = device #
OUTPUT:
V set if files open on the device

DESCRIPTION:
This is the complement to the CONNECT routine. The device will be removed from the hierarchy.

System Subroutines for Device Drivers

The following subroutines are available for device drivers. They should be used carefully as misuse may lock up the system. Equates for priorities and kernel services may be found in "kerequ.h".

NAME: DIVE - Prevent preemption

DESCRIPTION:
Whenever a device handler must access some critical resource like a buffer table and there is a possible race condition with another invocation of that driver, the DIVE call should be made. DIVE will prevent any other process from being scheduled. Note that DIVE does not inhibit interrupts.

NAME: SURFACE - Allow preemption

DESCRIPTION:
This call should be made after a DIVE, when there is no longer danger of a race condition. Calls to DIVE and SURFACE may be nested, thus it is safe to call a subroutine while "submerged" that uses DIVE and SURFACE.

NAME: KSVC - Kernel services

INPUT:
A = kernel service code
OUTPUT:
V set if error
A = error code

DESCRIPTION:
This is the entry point for requests that go to the kernel. The seven possible kernel services are: Clone this process, kill this process, P on (PØ), V on (PØ), install interrupt handler (PØ), ripoff interrupt handler (PØ), and reschedule. These values are defined in "kerequ.h".

NAME: APL - Advance priority level

INPUT:
A = new priority

DESCRIPTION:
This call is used to temporarily advance the priority level of a process while waiting for I/O. The priority of a process must always be at the minimum I/O level (defined in "kerequ.h") whenever it has possesion of system resources - a process cannot be killed at or above that level. APL leaves the old priority pushed on the user stack.

NAME: RPL - Restore priority level

DESCRIPTION:
RPL recalls a process's old priority from the user stack. If the old priority is below the I/O minimum and there is a pending kill on the process, the kill will then be completed.

NAME: SPL - Set priority level

INPUT:
A = new priority

DESCRIPTION:
This call sets a process's priority directly.

Writing a QUAD3 Interrupt Handler

Any hardware device may be added to the QUAD3 system and it may use the IRQ interrupt line to indicate a request for service. Since there are no vectored interrupts on the 6502, every interrupt is caught by the kernel. The kernel then polls for the source of the interrupt by calling each installed interrupt handler until one indicates that it has serviced the interrupt.

An interrupt handler can be user written and dynamically loaded. It is declared to the kernel with the INSTALL option of the KSVC call, with P0 pointing to the beginning of the subroutine. It must be removed with the RIPOFF option of the KSVC call with P0 again the address. Naturally, if the interrupt handler is in the user memory, care should be taken to remove it before all processes in the code body are defunct.

An interrupt handler has a very restricted environment. It may push data on the hardware stack, but it has no software stack. It may not call any system subroutines. The only service available is a V operation on the semaphore of its choosing.

NAME: Interrupt Handler

OUTPUT:

Z set if this interrupt now serviced
V set if V operation requested (Z must be set)
ksemptr -> semaphore to do V on

DESCRIPTION:

The interrupt handler must do an RTS like a normal subroutine. The kernel variable "ksemptr" is available if a V operation is to be done, just load it and set the V and Z bits on exit.

APPENDIX B

This is a listing of the macros for the 'W' machine. It was through the use of these macros that errors in repetitive coding sequences were virtually eliminated, speeding the debugging of the QUAD3 operating system.

```
/* A standard library of macros for the W machine */
/* Created 3-22-81 by Roger F. Powell */
/* Last edited 4-19-81 */

/******************** MOVES **********************/

    /* standard 2 byte move */
    MACRO MOVE src dst {
       if < $src.:Ø:1 = # > {
           if < $src = #Ø > [
               lda #Ø
               sta $dst
               sta $dst.+
               ]
           else    [
               lda $src.^
               sta $dst
               lda $src.!
               sta $dst.+
               ]
           }
       else    [
           lda $src
           sta $dst
           lda $src.+
           sta $dst.+
           ]
       }
```

```
/* move address */
MACRO MOVEA src dst {
   if < $src.:0:1 = # > {
       if < $src = #0 > [
           lda #0
           sta $dst
           sta $dst.+
           ]
       else    [
           lda $src.!
           sta $dst
           lda $src.^
           sta $dst.+
           ]
       }
   else    [
       MOVE $src $dst
       ]
   }

/* move byte */
MACRO MOVEB src dst [
   lda $src
   sta $dst
   ]

/* join two bytes at a destination word */
MACRO MOVEJ byte1 byte2 dst [
   lda $byte1
   sta $dst
   lda $byte2
   sta $dst.+
   ]

/* split a word to 2 destination bytes */
MACRO MOVES src byte1 byte2 [
   lda $src
   sta $byte1
   lda $src.+
   sta $byte2
   ]
```

```
/******************** COMPARES **********************/

/* standard 2 byte compare */
MACRO CMP src dst {
   if < $src.:Ø:1 = # > { set Sh $src.^ set Sl $src.! }
   else { set Sh $src set Sl $src.+ }
   if < $dst.:Ø:1 = # > { set Dh $dst.^ set Dl $dst.! }
   else { set Dh $dst set Dl $dst.+ }
   gdef    LCTR    Ø
   incr    LCTR
       [
       lda $Sh
       cmp $Dh
       bne L$LCTR
       lda $Sl
       cmp $Dl
   L$LCTR.:
       ]
   }

/* compare address */
MACRO CMPA src dst {
   if < $src.:Ø:1 = # > { set Sh $src.^ set Sl $src.! }
   else { set Sh $src.+ set Sl $src }
   if < $dst.:Ø:1 = # > { set Dh $dst.^ set Dl $dst.! }
   else { set Dh $dst.+ set Dl $dst }
   gdef    LCTR    Ø
   incr    LCTR
       [
       lda $Sh
       cmp $Dh
       bne L$LCTR
       lda $Sl
       cmp $Dl
   L$LCTR.:
       ]
   }

/* compare byte */
MACRO CMPB src dst [
   lda $src
   cmp $dst
   ]
```

```
/* compare two separate bytes with a destination word */
MACRO CMPJ bytel byte2 dst {
   if < $dst.:Ø:1 = # > { set Dh $dst.^ set Dl $dst.↓ }
   else { set Dh $dst set Dl $dst.+ }
   gdef    LCTR    Ø
   incr    LCTR
       [
       lda $bytel
       cmp $Dh
       bne L$LCTR
       lda $byte2
       cmp $Dl
   L$LCTR.:
       ]
   }

/* compare a word with two separate destination bytes */
MACRO CMPS src bytel byte2 {
   if < $src.:Ø:1 = # > { set Sh $src.^ set Sl $src.↓ }
   else { set Sh $src set Sl $src.+ }
   gdef    LCTR    Ø
   incr    LCTR
       [
       lda $Sh
       cmp $bytel
       bne L$LCTR
       lda $Sl
       cmp $byte2
   L$LCTR.:
       ]
   }

/************ OTHERS ************/

/* Call the kernel */
MACRO SCALL type [
   lda #$type
   jsr KSVC
   ]
```

```
/* Clear an area of memory up to 255 bytes long */
MACRO CLR start length {
   gdef LCTR Ø
   incr LCTR
        [
        ldy #Ø
        tya
   L$LCTR.: sta    Y,$start
        iny
        cpy $length
        blt L$LCTR
        ]
   }

/* End of maclib */
```

GLOSSARY

Various terms are defined here that are used throughout the document. No attempt has been made to define basic computer terms such as "RAM" or "compiler"; however, definitions of certain terms that are in standard use in the fields of Electrical Engineering and Computer Science have been presented here, as well as terms used exclusively in this document.

6502
The NMOS microprocessor that QUAD3 was developed on.

68000
An advanced (compared to the 6502) HMOS microprocessor.

Absolute symbol
A symbol that an assembler or linker will not modify based on any load address changes.

Background
Refers to programs that run on a computer that do not prevent the computer user from running another program right away.

Breakpoint
A debugging feature whereby a program can be stopped at some point during its execution in order for memory or CPU registers to be inspected.

Buffer
An area of memory where data is stored, prior to its being moved to another location or to a hardware device.

CHECKD
The QUAD2 file system integrity checker.

Child process
When a process in a multi-tasking environment creates another process, the new process is called the child while the old process is called the parent.

Context-switching
The act of switching control of the CPU from one process to another in a multi-tasking system.

Daemon
A system process that is created at system start-up in order to perform useful functions from time to time.

Device driver
The machine code that controls a hardware device. The term "low-level driver" is often used to describe those parts that poke the hardware, while the term "high-level" driver is used to describe those parts that implement a file system.

Device-independent I/O
The principle of interchangeability between files and hardware devices as far as programs are concerned. For example, any program that can write output to a disk file could unknowingly write output to a line printer if the special name of the line printer were given as the file name.

EIO
QUAD3's Extended I/O system. This is the interface that provides device-independent I/O by masking device driver characteristics from the programs.

External reference
A symbol in an assembly that refers to a value not known at assembly time because it is defined in another assembly.

File
A logical collection of data on a hardware device. A file may be viewed as a linear entity, regardless of the physical distribution of the data on the actual device.

File descriptor
A number provided by EIO to a process that opens a file, to be used in all further actions regarding the file.

Foreground
A program running in the foreground prevents the user from starting another program until this one is done. Most computer commands are like this.

General
The GOLEM firmware. This ROM provides basic memory display and the abililty to bootstrap various operating systems.

GOLEM
General Omnipotent Lugubriously Encephalic Microsystem. The physical hardware upon which this implementation of QUAD3 was done.

Hardware stack
The second page of memory in the 6502. Interrupts save the program counter and status register in this area.

I/O driver
See Device driver.

In-line file
A feature in LIZ where a short temporary file of data can actually be typed in as part of a command line.

Kernel
The heart of a multi-tasking system. The kernel is by definition indivisible, and thus it must contain the synchronization primitives and the scheduler.

Keyboard Interrupt
A certain key that when struck, will instruct the system to abort a foreground program.

LED
Light-emitting diode.

Linked-list
A set of buffers in memory, or sectors on a disk, where each element has a pointer somewhere within it that points to the next element.

Linker
A program that takes the output of several relocating assemblies and combines it into one file, with all of the external references correctly resolved.

LIZ
Short for "Lizard monitor". The QUAD3 command interpreter.

Load module
A file that is capable of being loaded into RAM and executed as a QUAD3 program.

Loader
That portion of QUAD3 that reads a load module into memory, fixing all of the relocatable bytes.

MACP
The macroprocessor developed for QUAD3.

Macro
An abbreviation for one or more assembly instructions. A macroprocessor takes a file that may have macros in it and "expands" each macro into the assembly instructions it represents.

Macroprocessor
Refer to Macro.

Mainframe
A large CPU with plently of memory and high-speed I/O devices.

Mini-shell
The debugging program designed to help develop QUAD3, that is now the "debug" command under LIZ.

Multi-tasking
Running more than one process at once. If one process is waiting (maybe for keyboard input), another process can use the CPU. Separate programs are often referred to as processes, not always correctly. Refer to Process.

P and V
Dijkstra's (1968) synchronization primitives, used in QUAD3.

Page
In the 65Ø2, a page of memory is 256 bytes.

Parent process
See Child process.

Pipes
FIFO (first-in first-out) buffers in memory that are used to connect the output of one program to the input of another as if they were files being read or written.

Pointer
An address.

Preemption
The act of taking away control of the CPU from a process without its explicit consent.

Priority
A number that determines a process's place in the "pecking order" when access to the CPU is arbitrated (scheduling).

Process
Strictly speaking, an abstract entity identified by the fact that it has a particular hardware and user stack and CPU state. One program can actually have several processes running using the same code.

Process ID
A unique number that identifies a particular process to the system.

QUAD2
The operating system developed for the GOLEM hardware before QUAD3. The name stems from an older system referred to as "QUAD1" or just "QUAD" - the letters stood for "QUick And Dirty". It was suggested that for QUAD2, they might be "QUaint And Dutiful".

QUAD3
The multi-tasking operating system designed and implemented for this thesis project.

Race
A situation where a result or action is indeterminate because it depends on the order of execution of two processes where such order is not guaranteed.

Raw I/O
Input or output without character translation or file system formatting.

Re-entrancy
The ability of some machine code to have more than process executing the code at the same time, without a race.

Relative symbol
A symbol during an assembly or link, whose value changes based on the load address.

Relocatable code
Machine code with enough extra information so that a loader may modify it based on where it is loaded in memory.

Relocating assembler
An assembler that generates relocatable code.

Scheduling
The act of allocating the CPU in a multi-tasking system.

Semaphore
A memory variable managed by the P and V synchronization primitives.

Sexton
A daemon process responsible for cleaning up after dead processes and notifying anyone who is interested in the death of the process.

Single-tasking
The inability to run more than one process at the same time.

Standard input/output/error output
The default file descriptors for programs that run under LIZ, for input, output, and diagnostic output (error messages).

Symbolic Parameter
An argument to a macro that allows changing the expansion based on information given on a per-expansion basis.

UART
Universal Asynchronous Receiver-Transmitter. A hardware device that is used for serial input and output.

Unified I/O
See Device-independent I/O

UNIX
A much-admired operating system for the PDP-11 and other mini-computers. UNIX is a trademark of Bell Labs.

User stack
An area of memory set up in QUAD3 to enable saving more data than is possible on the limited hardware stack.

Virtual device
A program that acts as if it were a hardware device of some sort.

VTF
Virtual Terminal Facility. This is the part of LIZ that processes raw terminal I/O and allows character and line deletion.

W machine

The expanded 6502 instruction set (done with macros) used to develop QUAD3

LIST OF REFERENCES

Bayer, D.L. and Lycklama, H., The MERT Operating System, Bell System Technical Journal (Jul-Aug 1978), pp 2049-2086.

Crowley, C., The Design and Implementation of a New UNIX Kernel, University of New Mexico Computer Science Report, September 1980.

Dijkstra, E.W., Cooperating Sequential Processes, in Programming Languages, ed. F. Genuys, New York: Academic Press (1968), pp 43-112.

Hoare, C.A., Monitors: An Operating System Structuring Concept, Communications of the ACM (Oct 1974), pp 549-557.

Jammel, A.J. and Stiegler, H.G., Managers versus Monitors, Proceedings of the 1977 IFIP Congress, ed. B. Gilchrist, pp 827-830.

Kernighan, B. and Plauger, P.J., Software Tools in Pascal, Addison-Wesley, Reading MA, 1980.

Knuth, D.E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, 2nd ed., Addison-Wesley, Reading MA, 1973.

Ritchie, D.M., and Thompson, K., The UNIX Time-Sharing System, Communications of the ACM (Jul 1974), pp 365-375.

Thompson, K., UNIX Implementation, Bell System Technical Journal (Jul-Aug 1978), pp 1931-1946.

Williams, T.B., Software Design Using Macroprocessors, Proceedings of the First Annual Phoenix Conference on Computers and Communications (May 1982), pp 125-130.